**Universidade de Brasília**
**Instituto de Ciências Exatas**
**Departamento de Ciência da Computação**

# Um Componente OpenStack para a Plataforma de Federação de Nuvens BioNimbuZ 2

Kilmer Luiz Lima Aleluia

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Aletéia Patrícia Favacho de Araújo

Brasília
2019

**Universidade de Brasília**
**Instituto de Ciências Exatas**
**Departamento de Ciência da Computação**

# Um Componente OpenStack para a Plataforma de Federação de Nuvens BioNimbuZ 2

Kilmer Luiz Lima Aleluia

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Aletéia Patrícia Favacho de Araújo (Orientadora)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. MSc. Edward Ribeiro
Senado Federal

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 12 de julho de 2019

# Dedicatória

Eu dedico este trabalho primeiramente aos meus pais, que nunca mediram esforços para que eu tivesse sempre a melhor educação possível. Dedico também à minha namorada, meus irmãos e aos meus amigos, com os quais compartilhei momentos de tristeza e felicidade, e também tiveram que ouvir tantas vezes a frase "depois que eu terminar o TG".

# Agradecimentos

Agradeço à Deus, meus pais e irmãos pelas oportunidades que me deram. À Aletéia por ter me aceitado como orientando e também por ajudar a encontrar os caminhos do meu projeto. À um grupo de orientandos da professora Aletéia que me auxiliaram e gentilmente usaram seu tempo me ajudando, em especial o Felipe e o Nides. Ao Ian por compreender minha situação e me dar oportunidade de me dedicar à UnB quando necessário. À minha namorada Débora por se preocupar comigo, sentir minha falta, sempre me ouvir e não me deixar desistir dos meus sonhos. Aos meus amigos, em especial da Central, da CJR, e do *Master Group*, por compartilharem conhecimento e me incentivar.

# Resumo

À medida que as tecnologias e as aplicações vão evoluindo, evoluem e mudam também as necessidades dos usuários. Assim, sob esse constante cenário, nasce a computação em nuvem, na qual os recursos são utilizados de acordo com a necessidade do cliente, de forma dinâmica, escalável e sob demanda. Com a grande quantidade de plataformas de serviços de computação em nuvem (como *Amazon Web Services*, *Google Cloud Plataform* e *Microsoft Azure*), surge o BioNimbuZ 2, uma plataforma de federação de nuvens orientada a microsserviços capaz de executar *workflows* de diferentes áreas científicas. Assim, para garantir uma total integração entre nuvens públicas, privadas, comunitárias e híbridas, notou-se a necessidade de desenvolver um componente na Camada de Federação do BioNimbuZ 2 para que o mesmo garanta uma total transparência na integraçao entre os diversos tipos de nuvem. O *software OpenStack* foi escolhido por se utilizar de *APIs RESTful* com boa documentação e também por ser um dos mais utilizado pela indústria.

**Palavras-chave:** BioNimbuZ 2, *OpenStack*, , Nuvem Computacional, Federação de Nuvens, Microsserviço.

# Abstract

As technologies and applications evolve, they evolve and change the needs of users as well. Thus, under this scenario, cloud computing is born, in which resources are used according to the customer's needs in a dynamic and scalable way and on demand. With the large number of cloud computing service platforms (such as Amazon Web Services, Google Cloud Platform and Microsoft Azure), *BioNimbuZ* 2 emerges, a microservice-oriented cloud federation platform capable of performing workflows from different scientific areas. Therefore, to ensure a complete integration between public, private, community and hybrid clouds, it was noted the need to develop a component in the *BioNimbuZ* 2 Federation Layer so that it can guarantee a total transparency in the integration between the different cloud types. OpenStack was chosen because it is a RESTful API software with a good documentation and is one of the most used by the industry.

**Keywords:** BioNimbuZ 2, OpenStack, Cloud Computing, Cloud Federation, Microservice.

# Sumário

# Lista de Figuras

# Capítulo 1

# Introdução

Com o passar dos anos, a demanda computacional pela indústria e pela academia apresentou um aumento gradativo. Assim, servidores locais e *datacenters* podem não ser suficientes para suprir determinadas demandas, mas também podem, por outro lado, possuir mais recursos que o necessário, acarretando em gasto de recursos e de energia. Sob tal contexto, surge a computação em nuvem, que tem como uma das principais características o *pay-per-use* [2], que garante que recursos sejam alocados de acordo com a demanda e, como consequência, paga-se apenas pelos recursos consumidos.

Além da característica citada, outras características da computação em nuvem são a elasticidade, a interoperabilidade e a capacidade de adaptação automática [2]. Uma grande vantagem é que tal modelo possibilita acesso aos recursos por intermédio da Internet [3], não mais utilizando espaços em ambientes comerciais, nos quais o valor do metro quadrado pode representar uma alta despesa.

Por outro lado, muitas vezes as nuvens dão ao usuário a falsa impressão de possuírem recursos ilimitados. Contudo, por maior que seja a capacidade tecnológica de um provedor de nuvem, sempre é possível que uma aplicação esgote os recursos disponíveis caso exista uma demanda muito alta. Assim, surge a ideia de um novo modelo, chamado de federação de nuvens [4]. Federação de nuvens é um conjunto de nuvens integradas que são controlados por intermédio de uma única interface, aproximando-se da ideia de "recursos ilimitados". Isso é possível pois uma vez que o limite de uma nuvem seja atingido, outra nuvem pode ser adicionada, e assim por diante. Há na literatura diversas propostas de arquiteturas para nuvens federadas [5, 6, 7, 8, 9, 10].

O paradigma de federação de nuvens propõe que diversos provedores estabeleçam acordos com a finalidade de ampliar os recursos disponíveis para o usuário. Dessa maneira, as plataformas de nuvens federadas devem mostrar aos clientes uma independência de provedores. Os usuários, portanto, não devem se preocupar nem precisam saber os detalhes da implementação de cada um dos provedores da plataforma, o que transparece a ideia

de apenas um grande provedor de recursos. Nesse cenário, surge a plataforma de federações de nuvens BioNimbuZ [11, 12, 13, 14, 15, 16, 17], que implementa uma solução com arquitetura monolítica de nuvens federadas para a execução de *workflows* científicos de alto poder computacional. Um *workflow* é uma sequência de tarefas a serem executadas, compostas por um conjunto de dados de entrada e de saída, cuja execução é ordenada com a finalidade de alcançar um objetivo [18, 19].

Havia, entretanto, algumas limitações no BioNimbuZ, ligadas principalmente à natureza monolítica da plataforma, pois ela foi desenvolvida inicialmente sob uma única aplicação, mesmo que ainda tivessem diversas camadas envolvidas. Assim, alterações do sistema que seriam pequenas tornaram-se grandes devido ao alto grau de acoplamento do sistema monolítico.

A partir desse cenário, foi desenvolvido o BioNimbuZ 2 [1], que é uma evolução da sua primeira versão, e foca em uma abordagem em microsserviços, a partir de uma arquitetura paralela e distribuída. Para a comunicação entre a plataforma e os provedores foram implementados componentes (anteriormente chamados de *plugins*) para *Amazon Web Services* [20] e também para o *Google Cloud Platform* [21].

Entretanto, faz-se necessária a integração com provedores privados que sejam gerenciados pelo *OpenStack* [22], que são sistemas que permitem a criação de nuvens tanto públicas quanto privadas. Diante do exposto, este trabalho propõe implementar um componente para o BioNimbuZ 2 que faça a integração entre os provedores públicos e privados, garantindo uma federação de nuvens capaz de interagir com qualquer tipo de nuvem.

## 1.1 Motivação

O BioNimbuZ 2 tem se mostrado uma eficiente ferramenta para integrar provedores públicos de nuvem. Todavia, ele não tem nenhum componente para a integração com uma nuvem privada, como o *OpenStack*.

Assim, a motivação deste trabalho é gerar uma integração entre a plataforma BioNimbuZ 2 e uma nuvem privada construída por intermédio do *OpenStack*, garantido uma federação de nuvens híbrida, controlada e gerenciada pelo BioNimbuZ 2.

## 1.2 Problema

Desenvolver um componente para a plataforma BioNimbuZ 2 que garanta a associação entre nuvens pública e privadas.

## 1.3 Objetivos

O objetivo geral deste trabalho é o desenvolvimento de um componente que vincule uma nuvem privada *OpenStack* com a plataforma BioNimbuZ 2. Para que o objetivo geral seja cumprido, faz-se necessário atingir os seguintes objetivos específicos:

- Analisar as características que um componente do BioNimbuZ 2 deve possuir;
- Definir os *endpoints* disponibilizados pelas *API's OpenStack* a serem utilizados;
- Propor um componente que seja capaz de se comunicar com nuvens privadas do *OpenStack*;
- Integrar o novo componente ao BioNimbuZ 2;
- Avaliar a execução de *workflows* em uma nuvem gerenciada pelo componente *OpenStack* proposto.

## 1.4 Estrutura do Trabalho

Além deste capítulo de introdução, este trabalho possui ainda outros seis capítulos. No próximo capítulo será feita uma explicação geral sobre os temas computação em nuvem e nuvens federadas.

No Capítulo 3 será explicado o funcionamento e a arquitetura do BioNimbuZ 2. Nesse capítulo também são apresentados trabalhos relacionados ao universo de nuvens federadas.

O Capítulo 4 possui explicações sobre o *OpenStack*, a ferramenta escolhida para gerenciar as nuvens privadas. Em seguida, os detalhes sobre a implementação e os testes do componente estão no Capítulo 5.

Por fim, no Capítulo 6, as conclusões pertinentes deste trabalho e as sugestões para trabalhos futuros são apresentadas.

# Capítulo 2

# Referencial Teórico

Neste capítulo, nas Seções 2.1 e 2.2, serão apresentadas uma descrição e uma revisão sobre os principais temas desta monografia. Por fim, a Seção 2.3 discorre acerca de microsserviços, que é uma tecnologia utilizada na implementação do componente proposto neste trabalho.

## 2.1 Computação em Nuvem

O modelo de computação em nuvem tem como premissa a capacidade dos usuários pagarem apenas pelo que foi utilizado, e esses recursos são fornecidos sob demanda pela Internet, permitindo poder de processamento, de armazenamento e também de rede [23].

A literatura apresenta diferentes definições para ambiente de nuvens. Algumas delas são:

- Armbrust *et al.* [24] definem como *"a união de aplicações oferecidas como serviço pela Internet, com o hardware e o software localizados em datacenters de onde o serviço é provido"*;
- Para Buyya *et al.* [23], "*Nuvem é um sistema de computação paralela e distribuída que consiste em um conjunto de computadores interligados e virtualizados que são dinamicamente providos e apresentados como um ou mais recursos de computação unificada baseada em contratos de níveis de serviço estabelecidos através de negociação entre o prestador de serviço e os consumidores*";
- *Mell et al.* [3], definem como "*A computação em nuvem é um modelo para acesso conveniente, sob demanda, e de qualquer lugar, a uma rede compartilhada de recursos de computação (isto é, redes, servidores, armazenamento, aplicativos e serviços) que possam ser prontamente disponibilizados e liberados com um esforço mínimo de gestão ou de interação com o provedor de serviços*".

Como pode ser observado, não existe um consenso em relação à definição de computação em nuvem. Todavia, Vaquero *et al.* [2], a partir de uma avaliação com mais de 20 definições, sugeriram uma proposta genérica, em que a computação em nuvem é *"um grande conjunto de recursos virtualizados e compartilhados (hardware, plataformas de desenvolvimento ou software) que podem ser facilmente acessados e utilizados. Esses recursos podem ser dinamicamente reconfigurados para se ajustarem a uma carga variável de trabalho, permitindo uso otimizado dos mesmos. Este conjunto de recursos é tipicamente explorado por um modelo de pagamento por utilização chamado de pay-per-use em que as garantias são oferecidas pelo provedor de infraestrutura por meio de contratos de serviço personalizados (Service Level Agreement– SLA)".*

Diante do exposto, neste trabalho será considerada a definição proposta por Vaquero *et al.* [2].

### 2.1.1 Características

A computação em nuvem tem algumas importantes características, enumeradas por *Badger et al.* [25], dentre essas, as essenciais são:

- **Auto-atendimento sob demanda (*On-Demand Self-Service*)**: é a característica que trata da possibilidade do cliente usar os recursos quaisquer oferecidas pela nuvem, como armazenamento e processamento, de acordo com suas necessidades quando ele mesmo desejar, sem necessidade de um terceiro, como administrador do provedor da nuvem;
- **Amplo acesso à rede (*Ubiquitous Network Access*)**: é possível acessar qualquer serviço da nuvem desde que o dispositivo possua acesso à Internet;
- **Elasticidade (*Elasticity)***: a elasticidade trata da capacidade que a nuvem deve ter de aumentar ou diminuir com o menor tempo dispendido possível os recursos alocados, a fim de otimizar os recursos do usuário;
- ***Pool* de Recursos (*Resource Pooling*)**: sem a necessidade de saber onde os recursos fisicamente estão, eles são apresentados aos consumidores do provedor como um grande *pool*. Em geral, é apresentado ao usuário apenas informação mais ampla, como o país onde encontra-se o *data center*;
- **Serviços Mensuráveis (*Measured Service*)**: os recursos e serviços da nuvem podem ser monitorados e manipulados pelo usuário ou pela nuvem, automaticamente. Isso habilita a capacidade do fornecedor da nuvem cobrar do consumidor exatamente o que foi utilizado, ou até permitir que o cliente defina um limite superior de recursos que se encaixe no seu limite orçamentário.

### 2.1.2 Modelos de Serviço

Em relação ao paradigma de computação em nuvem, existem diferentes modelos de serviços, os quais são baseados em suas características. Para *Mell et al.* [3], pode-se classificar tais modelos em três diferentes tipos:

- **IaaS (*Infrastructure-as-a-Service*)**: pode ser considerado o tipo de serviço mais fundamental da computação em nuvem, uma vez que abrange os recursos de armazenamento, rede e computação necessários para as organizações, sob a qual é possível executar qualquer *software*, tal como sistemas operacionais ou aplicativos;
- **PaaS (*Platform-as-a-Service*)**: engloba os serviços essenciais para o desenvolvimento de sistemas e, por isso, geralmente é utilizado por desenvolvedores, a fim de possuir um ambiente de desenvolvimento, teste e gerenciamento de uma aplicação;
- **SaaS (*Software-as-a-Service)***: serviço de fornecimento de aplicativo que é executado em infraestrutura na nuvem, mas sem a necessidade do consumidor conhecer ou gerenciar tal infraestrutura.

### 2.1.3 Modelos de Implantação

Os modelos de implantação classificam nuvens de acordo com a amplitude de acesso dos envolvidos. *Mell et al.* [3] definem tais modelos como sendo:

- **Nuvem Privada**: apenas uma organização tem acesso a uma infraestrutura de nuvem. É possível que a organização seja proprietária dos recursos físicos. O gerenciamento pode ser realizado pela própria empresa ou por terceiros;
- **Nuvem Comunitária**: diversas organizações com interesses similares (como, por exemplo, requisitos de segurança) compartilham a infraestrutura da nuvem. O gerenciamento pode ser feito por uma ou várias organizações ou também por terceiros;
- **Nuvem Pública**: a infraestrutura é aberta para qualquer organização ou usuário interessados, ou seja, qualquer um desses é capaz de acessar e provisionar recursos da nuvem em questão. Pode ser o caso de organizações acadêmicas, comercias ou do governo;
- **Nuvem Híbrida**: trata de infraestruturas compostas, pelo menos, por duas das modalidades supracitadas.

## 2.2 Nuvens Federadas

Até aqui foram listados vários benefícios e privilégios da computação em nuvem. Todavia, em alguns casos, ela não é suficiente para suprir as necessidades de aplicações de usuários e consumidores. *Armbrust et al.* [24] citaram alguns empecilhos ou limitações, como a disponibilidade do serviço e a continuidade do negócio.

Nesse contexto, surge a federação de nuvens, um modelo computacional que permite que diferentes provedores de nuvem possam se integrar com outras nuvens e, então, melhorarem seus recursos e suas capacidades. Segundo *Puliafito et al.* [6], uma federação de nuvens é um sistema no qual dois ou mais provedores de serviço de computação em nuvem cooperam. Essa cooperação é importante para eliminar ou minimizar os empecilhos e as limitações da computação em nuvem.

Diante do exposto, é importante ressaltar que alguns termos utilizados na literatura podem ser confusos ou até parecidos, por isso faz-se necessário definir claramente alguns desses termos, sendo que os principais neste trabalho são:

- **Federação de Nuvens**: sem que seja necessária a comunicação com um administrador, é possível que as nuvens (que participam de forma voluntária no estabelecimento de conexão) estendam e aumentem seus recursos de forma automática [26];
- ***Multi-Cloud***: neste caso há terceiros encarregados a gerenciar os recursos das diferentes nuvens envolvidas, o que significa, portanto, que nuvens não têm participação voluntária no estabelecimento de conexão com outras nuvens [26];
- ***Inter-Cloud***: trata, de maneira mais abrangente, de relações quaisquer entre nuvens, o que inclui as duas anteriores, estendendo os recursos das nuvens, gerando uma chamada "nuvem-de-nuvens" [26].

### 2.2.1 Fases da Evolução

Para descrever o processo de evolução do paradigma da computação em nuvem, *Bittman* [27] dividiu o modelo em três estágios:

1. **Monolítico**: é qualificado por grandes ilhas proprietárias, na qual grandes empresas fornecem os serviços, como Google [21], Amazon [20] e Microsoft [28];
2. **Cadeia Vertical de Suprimentos**: é uma primeira fase de integração entre as nuvens. Ainda existe um foco em ilhas proprietárias, mas outros provedores passam a fornecer os serviços;

3. **Federação Horizontal**: neste estágio passa a existir um nível de interoperabilidade entre diferentes provedores de nuvem. Assim sendo, podem existir diversos ganhos, como economia de escala, aumento de recursos e capacidades, ou até uso eficiente de seus ativos.

### 2.2.2 Vantagens das Federações de Nuvens

A utilização de federações de nuvens traz vários benefícios e resolvem problemas que outros paradigmas, como o próprio modelo de computação em nuvem, não conseguem solucionar. Dentre tais benefícios, destacam-se [26, 29]:

- **Disponibilidade**: com diversos provedores, tem-se, consequentemente, diversas regiões de nuvens ao redor do globo. Isso permite também a redundância de informações e permite servir o consumidor de acordo com a região mais próxima a ele, diminuindo, portanto, a latência;
- **Gastos controlados**: é possível escolher os provedores mais econômicos, tarefa que pode ser delegada ou à própria federação ou à empresa;
- **Independência do fornecedor**: o usuário pode escolher o provedor que desejar com a prioridade que desejar (preço, recursos, confiança), sendo permitido, também, trocar o provedor quando julgar necessário;
- **Melhor aproveitamento de recursos**: devido à elasticidade, não há subutilização de recursos, nem há problemas quando os recursos estão esgotados. Nesse caso, outros recursos serão alocados, caso isso esteja previsto no acordo entre cliente e federação. Por vezes, provedores de nuvem podem dar impressão de que os seus recursos são ilimitados, o que não é um fato. Esse é mais um caso em que as federações ajudam, pois torna muito mais improvável que uma aplicação esgote todos os recursos dos diferentes provedores disponíveis na federação, mesmo levando em consideração que a ideia de recursos ilimitados seja algo improvável.

### 2.2.3 Desafios das Federações de Nuvens

Com tantos provedores diferentes numa mesma plataforma, é natural imaginar que existam desafios e limitações para se manter uma federação de nuvens. Dentre eles, destacam-se [26, 30, 31]:

- **Dificuldade na manutenção do SLA**: uma vez que cada nuvem possui seus próprios Acordos de Nível de Serviço (*Service Level Agreement* - SLA), a tarefa de gerenciar e garantir a manutenção do SLA torna-se muito complexa;

- **Falta de padrões**: em diversas frentes diferentes, como segurança, máquinas virtuais, protocolos de mensagens, autenticação e armazenamento, existem diferentes abordagens de acordo com o provedor. Como consequência, dificulta o desenvolvimento de sistemas que utilizam diversos recursos de diferentes provedores de nuvens. Em alguns casos, isso pode gerar outro problema, chamado *vendor lock-in* [32], que é quando a aplicação não sabe lidar com a migração de uma tecnologia pra outra, então o cliente fica "preso" a uma tecnologia específica;
- **Monitoramento complexo**: se a federação tiver uma grande quantidade de nuvens e de provedores, é possível que tais nuvens estejam em diversos lugares do Mundo. Isso causa latência na rede da comunicação, o que dificulta a consolidação dos dados de recursos da federação.

Diante dos desafios apresentados nesta seção, uma solução proposta nos últimos anos é o uso de microsserviços [33] nas implementações de plataformas de nuvens federadas. Os microsserviços serão discutidos em detalhes na Seção 2.3.

Desta forma, plataformas de nuvens federadas [4] funcionam como uma grande provedora de recursos, na qual os usuários não precisam saber detalhes acerca da implementação dos componentes de cada provedor específico. Assim, representam uma interface entre as diferentes nuvens disponíveis e o cliente.

## 2.3 Microsserviços

Os microsserviços são serviços com pequenas responsabilidades, que são executados de maneira independente, com seus próprios processos [33]. Idealmente, comunicam-se por intermédios de mecanismos como uma API REST sobre o HTTP - *Hypertext Transfer Protocol*. É amplamente utilizado por grandes empresas, como Airbnb [34], Netflix [35], SoundCloud [36] e Uber [37], que decidiram se atualizar de uma natureza originalmente monolítica para um paradigma distribuído. A Figura 2.1 compara uma arquitetura orientada a microsserviços com uma monolítica.

Newman [38] pontua diversas vantagens de se utilizar sistemas com microsserviços, as quais são:

- **Alinhamento organizacional**: quanto maior o sistema, maior a dificuldade para se manter o alinhamento entre as equipes. O uso de microsserviços torna mais simples a distribuição de equipes, e elas tendem a diminuir, com menos desenvolvedores por equipe;
- **Compatibilidade**: com a utilização de microsserviços é mais simples aplicar técnicas de compatibilidade ou até mesmo responsividade, para uma maior reutilização

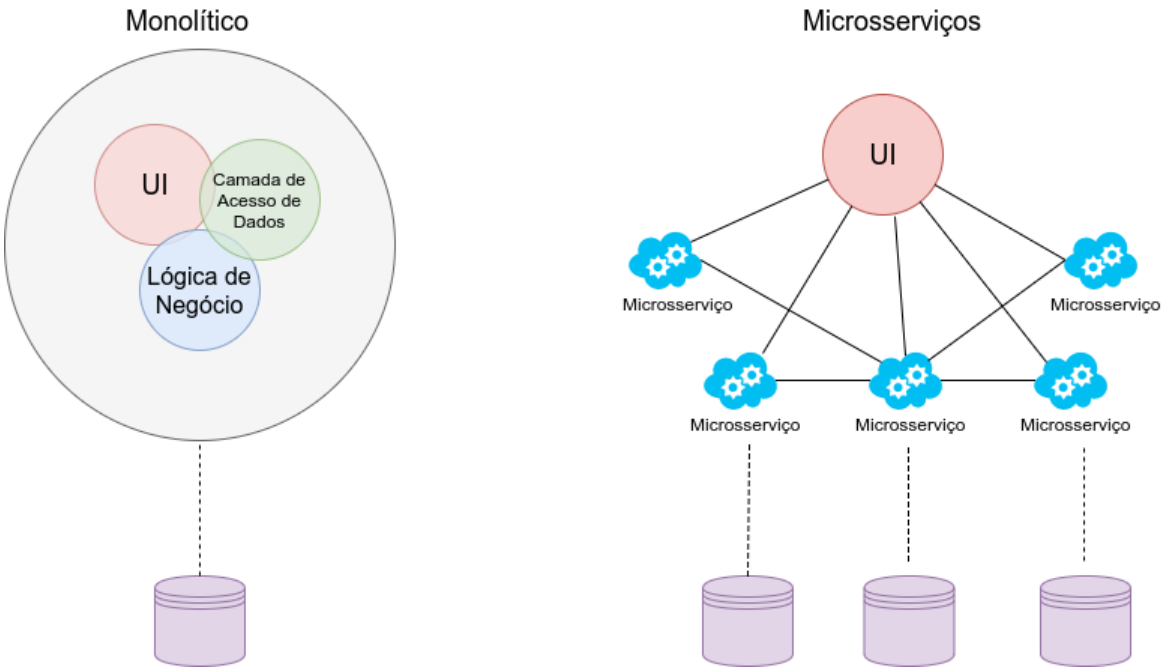


Figura 2.1: Arquitetura monolítica e orientada a microsserviços.

dos serviços, tal que o mesmo sistema seja capaz de ser reproduzido em diferentes plataformas, como aplicativo desktop, web ou até *mobile*;

- **Escalabilidade**: é consideravelmente mais simples aplicar técnicas de escalabilidade em um ambiente de microsserviços, uma vez que para alterar as propriedades do sistema, basta aumentar ou reduzir os recursos de forma independente em cada microsserviço de acordo com a demanda. No caso de sistemas monolíticos seria provavelmente necessário considerar e até atualizar todo o sistema;
- **Otimização em migrações**: é comum, principalmente, em sistemas muito grande e antigos, que existam nele tecnologias antigas e pouco usadas. Isso acarreta dificuldades na substituição dessas tecnologias. No contexto de microsserviços, tal migração é facilitada, uma vez que, como são independentes, pode-se atualizar aos poucos, serviço por serviço, de maneira segura e sem comprometer o resto do sistema;
- **Resiliência a falhas**: como os microsserviços possuem baixo acoplamento e são altamente independentes, quando ocorre a falha em um desses serviços, os demais não são afetados, tornando a falha isolada e, potencialmente, diminuindo os danos no sistema como um todo;
- **Tecnologias heterogêneas**: como o sistema é composto por serviços independentes que colaboram entre si, é possível que cada serviço possua sua tecnologia

específica de acordo com cada problema, otimizando o sistema.

## 2.4 Considerações Finais

Neste capítulo foram abordados os principais conceitos de computação em nuvem, dando ênfase as mais relevantes características, sua arquitetura e seus modelos de implantação. Outros conceitos abordados foram as principais características e as classificações da computação em nuvem. Por fim, foram apresentadas as definições de federação de nuvens. No próximo capítulo serão apresentadas as principais características da Plataforma de Federação BioNimbuZ 2.

# Capítulo 3

# Plataforma de Federação de Nuvens BioNimbuZ 2

BioNimbuZ 2 é uma plataforma de federação de nuvens sugerida por Mendes [1] que foi utilizada como base neste trabalho. Questões acerca de seus mecanismos e sua arquitetura hierárquica e distribuída orientada a microsserviços são detalhados nas seções a seguir.

Na Seção 3.1 é exposta uma visão geral sobre os componentes que compõem a federação. A Seção 3.2 explica as camadas que constituem a arquitetura da plataforma. Por fim, na Seção 3.3, as considerações finais sobre a plataforma BioNimbuZ 2 são apresentadas.

## 3.1 Visão Geral

O BioNimbuZ 2 é uma plataforma de federação de nuvens orientada a microsserviços [1]. A plataforma, que é bastante eficiente para execução de tarefas, é capaz de executar também aplicações genéricas, partilhar credenciais e espaços de armazenamento e efetuar o *download* ou *upload* direto de arquivos para os espaços de armazenamento dos usuários do sistema [1].

A arquitetura do BioNimbuZ é hierárquica e distribuída, a fim de reduzir o acoplamento e aumentar independência dos serviços e camadas. Para a associação entre a plataforma e os provedores de nuvem são implementados componentes (antes chamados de *plugins*), que são microsserviços e operam de maneira independente em relação ao funcionamento geral do sistema, o que significa que podem ser iniciados e parados a qualquer momento, contribuindo no tangente à tolerância a falhas.

Assim sendo, é importante citar a maneira que tais serviços se comunicam. Existem dois níveis para se tratar a comunicação entre os componentes, uma em alto nível e outra em baixo nível. O primeiro nível, em alto nível, as nuvens trocam informações entre si de maneira direta. No segundo caso, de comunicação em baixo nível, a troca de dados

acerca das tarefas em execução ocorre de maneira interna em cada nuvem que compõe a federação. Tais componentes são divididos em diferentes níveis ou camadas.

Para isso, a arquitetura do BioNimbuZ 2 é composta por quatro camadas [1]: a Camada de Aplicação, a Camada de Coordenação de Tarefas, a Camada de Federação e a Camada de Execução. A Figura 3.1 ajuda a visualizar tal arquitetura em camadas, e mostra como cada camada é composta por controladores e serviços, e são divididos a fim de se conquistar um bom aproveitamento de recursos, tanto de comunicação quanto de processamento e armazenamento.

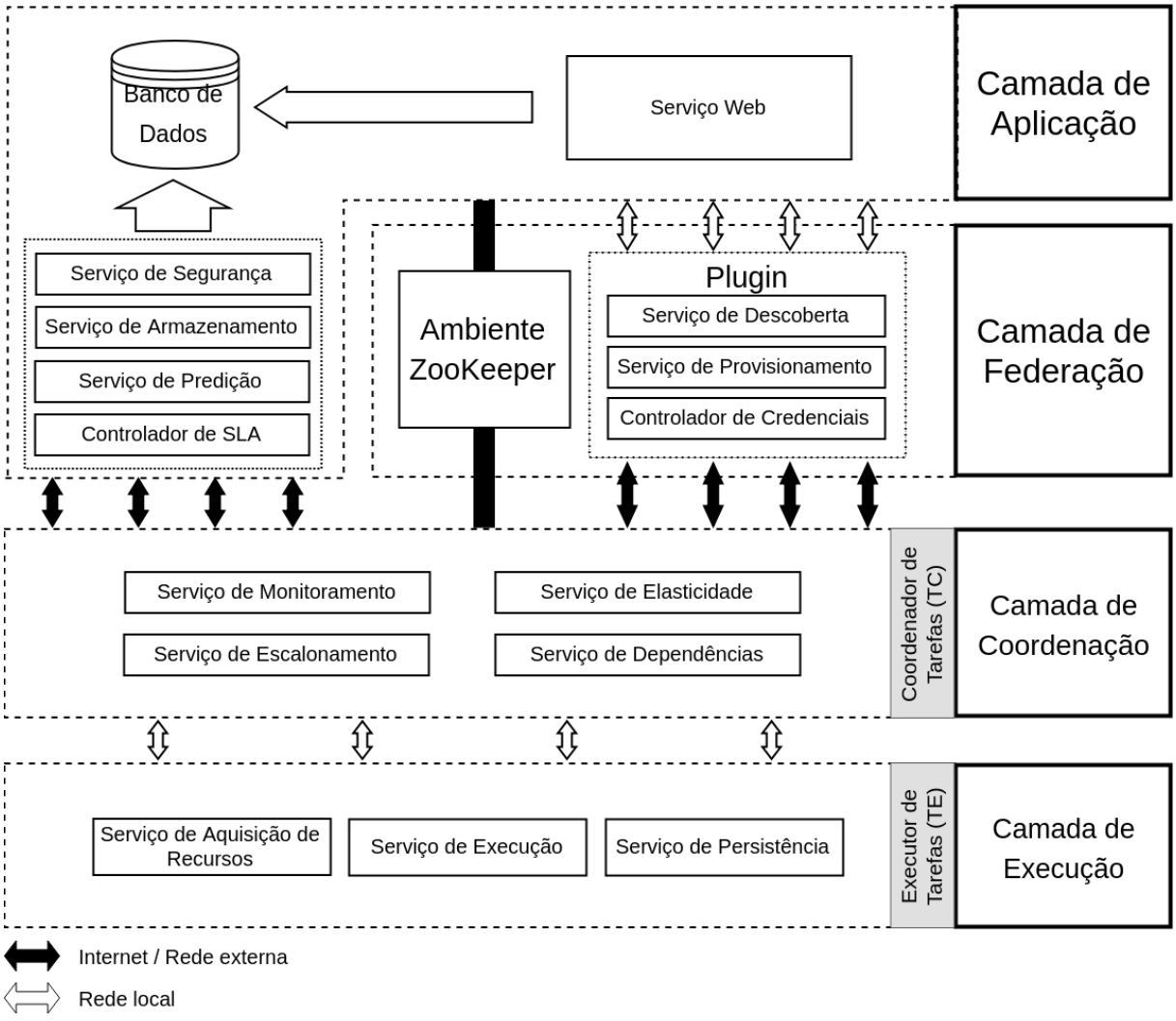


Figura 3.1: Arquitetura da plataforma BioNimbuZ 2 [1]

## 3.2 Arquitetura

Como dito anteriormente, a plataforma BioNimbuZ 2 possui quatro camadas e elas comunicam-se entre si para a execução de tarefas e *workflows* em ambiente federado, e também para o gerenciamento dos recursos. As camadas serão descritas a seguir, da Seção 3.2.1 a Seção 3.2.4.

### 3.2.1 Camada de Aplicação

É a camada de interface web da plataforma, que permite que os usuários utilizem o sistema. A partir desta camada é possível que o usuário consiga, por exemplo, salvar suas credenciais e compartilhar com grupos específicos. Isso permite que organizações, como equipes de desenvolvimento ou grupo de estudantes, compartilhem das mesmas credenciais. É na camada de aplicação que se torna possível que o usuário envie dados de entrada e de saída, ou criem as dependências de aplicações, como *workflows* científicos de Bioinformática. Os citados dados são salvos em um banco de dados PostgreSQL [39] na versão 9.5.

A Camada de Aplicação é composta por quatro serviços e um controlador, descritos a seguir:

- **Controlador de SLA:** o objetivo deste controlador é garantir que os acordos definidos pelos usuários envolvidos no SLA (*Service Level Agreement*) sejam cumpridos. Para isso, tal controlador examina frequentemente as informações levantadas pelos Coordenadores do BioNimbuZ 2, descritos na Seção 3.2.4;
- **Serviço de Armazenamento:** a partir de pontos importantes, como custo e capacidade, os próprios componentes e o Serviço de Descoberta levantam dados, a partir dos quais o Serviço de Armazenamento toma as decisões de armazenamento de arquivos de entrada e de saída que potencialmente surgem nos *workflows*;
- **Serviço de Predição:** é o serviço de predição para os *workflows* elaborados. Ajuda o cliente na escolha do ambiente computacional com estimativas de tempo, custo financeiro e recursos a serem utilizados. Neste serviço serão levados em consideração apenas os provedores que o usuário já possui cadastro na plataforma. Por exemplo, se o cliente possui credenciais do *Google Cloud Plataform* mas não possui para o *Amazon Web Services*, o serviço de predição apenas levará em consideração os recursos do *Google*;
- **Serviço de Segurança:** é o serviço que possui a responsabilidade acerca das credenciais e das autenticações dos provedores de nuvens disponíveis;

- **Serviço Web:** é o serviço que, efetivamente, garante a interação entre o usuário e a plataforma. É a camada na qual o usuário pode fornecer credenciais, elaborar *workflows*, definir dados de entrada e de saída, ou monitorar a execução de tais *workflows*.

### 3.2.2 Camada de Federação

A Camada de Federação tem como principal responsabilidade agregar os componentes do sistema independente de qual provedor de nuvem eles tenham origem, formalizando, assim, uma federação de nuvens.

Esta camada é principalmente composta pelo *ZooKeeper* [40] e pelos componentes de integração com nuvens, que são responsáveis por implementar as funcionalidades dos provedores de nuvem. É também a principal camada que sofrerá alterações com este trabalho, pois é necessário desenvolver nela um componente para o OpenStack.

*Apache ZooKeeper* [41] é uma ferramenta que tem como principal objetivo desenvolver e manter um servidor de código aberto, que viabilize uma coordenação distribuída altamente confiável. Assim, é possível distribuir os dados entre os Coordenadores e a Camada de Aplicação para, então, examiná-los e monitorá-los em tempo real. Os Coordenadores serão detalhados na Seção 3.2.3.

Tanto os componentes já implementados quanto o proposto neste trabalho, ambos funcionam como microsserviços. Todas as responsabilidades acerca de determinadas nuvens são direcionadas para serem tratadas pelo seu respectivo componente. Assim, cada um destes componentes possui controladores e serviços de diversos aspectos, como Controlador de Armazenamento, Controlador de Computação, Controlador de Credenciais, Controlador de Imagem, Controlador de Informações, Controlador de Precificação, Serviço de Descoberta e Serviço de Provisionamento. A comunicação entre esses serviços é redigida pelo estilo de arquitetura REST (*Representational State Transfer*) [42]. Detalhes sobre o componente para o *OpenStack* serão explicitados no Capítulo 5.

### 3.2.3 Camada de Coordenação

A Camada de Coordenação e seus respectivos serviços são implementados por Coordenadores de Tarefas, os TC's (*Task Coordinators*, ver Figura 3.2). Como o próprio nome sugere, os TCs têm como compromisso fiscalizar e coordenar o fluxo de tarefas, referente ao *workflow* do usuário, e também as dependências entre as próprias tarefas que estão sendo executadas. Um *workflow* é uma sequência de tarefas a serem executadas, compostas por um conjunto de dados de entrada e de saída, cuja execução é ordenada com a finalidade de alcançar um objetivo [19].

Quando um fluxo de tarefas definido pelo usuário se inicia, é criado um TC e, para cada tarefa, cria-se um TE, *Task Executor* – Executor de Tarefas. Tais executores verificam constantemente se as respectivas tarefas podem ou não ser executadas, ou se as tarefas devem ser encerradas, o que evita consumo equivocado de recursos da nuvem. O TC é gerado na própria nuvem destino, para que, assim, fique perto das tarefas e diminua, portanto, o tempo de comunicação para uma melhor performance do Coordenador. Ele é gerado utilizando também as credenciais do cliente, e, consequentemente, o custo do TC é adicionado ao valor final do usuário. De maneira geral, esse valor a mais é válido, uma vez que os benefícios gerados favorecem a execução segura e íntegra do *workflow*. A Figura 3.2 mostra detalhes acerca da arquitetura hierárquica e distribuída em conjunto com TCs e TEs.

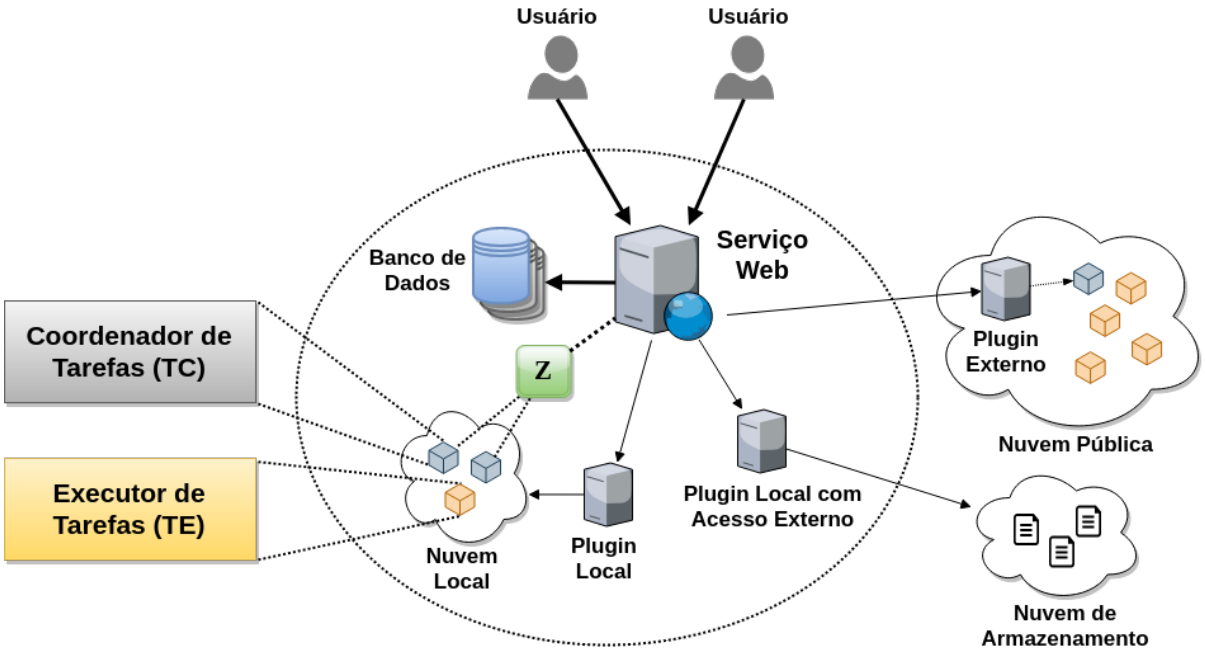


Figura 3.2: Distribuição dos componentes da arquitetura do BioNimbuZ 2.

A Figura 3.3 auxilia no entendimento do funcionamento do Coordenador e do Executor de Tarefas. No início do fluxo apresentado na Figura 3.3, existem quatro tarefas, que são as Tarefas 1, 2, 3 e 4, divididas entre as nuvens A, B e C, e são executadas em paralelo. No início do processo foi criado um TC para as nuvens A, B e C, e para cada tarefa é criado um TE. No caso da nuvem A, após o fim da execução da Tarefa 1, seu TC é encerrado, e torna a ser acionado apenas ao fim da Tarefa 6. No caso da Nuvem C, o TC é finalizado quando a Tarefa 4 acaba. Isso acontece pois uma tarefa pode demorar horas ou até dias para ser executada, e se o TC não fosse interrompido, a nuvem ficaria ociosa e gastaria recursos.

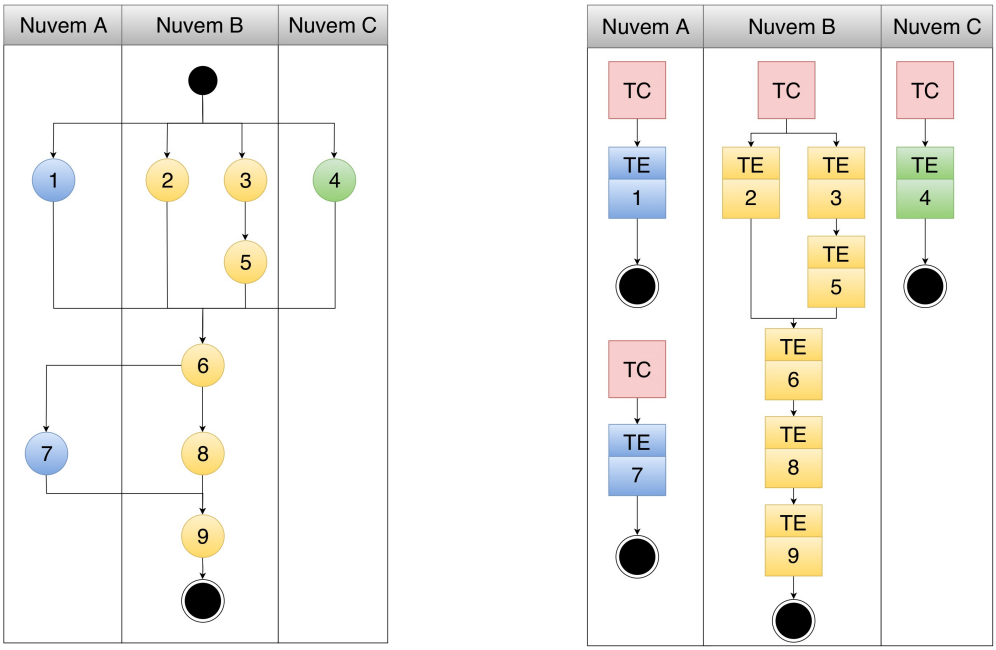


Figura 3.3: Fluxo de execução de tarefas na plataforma BioNimbuZ 2. À esquerda, execução percebida pelo usuário, e à direita o comportamento real da aplicação.

Além do que já foi explicitado, a Camada de Coordenação possui outros serviços importantes:

- **Serviço de Dependências:** esse serviço continuamente verifica dependências e demandas das tarefas, ou se existem próximas tarefas a serem realizadas. É também o serviço que indica se um certo conjunto de atividades foi encerrado ou não. Outra atividade desse serviço é verificar se é possível que o TC seja encerrado, caso convenha, a fim de economizar gastos;
- **Serviço de Elasticidade:** em conjunto com o Serviço de Monitoramento, examina os dados obtidos pelos TEs (*Task Executors*), e decide se é necessário diminuir ou aumentar recursos, tanto de maneira horizontal, por exemplo modificando quantidade de máquinas virtuais, quanto de maneira vertical, por exemplo alterando o tamanho da memória ou a capacidade da CPU;
- **Serviço de Escalonamento:** é o serviço responsável por distribuir as atividades a serem realizadas nas máquinas, além de solicitar o início de uma tarefa ou um lote de tarefas;

- **Serviço de Monitoramento:** tal serviço tem como responsabilidade estar regularmente coletando dados e informações acerca dos Executores de Tarefas, ou TEs. Esses dados auxiliam em outras atividades, como garantia dos SLA's ou até ajudando o Serviço de Elasticidade.

### 3.2.4 Camada de Execução

A Camada de Execução foi implementada utilizando-se da abordagem de microsserviços. É responsável pela execução das tarefas de um fluxo do usuário numa máquina virtual. Esta camada está constantemente em comunicação com o seu respectivo Coordenador, para informar dados sobre as execuções.

Os *Task Executors*, ou Executores de Tarefas, são processos que compõem esta camada, e utilizam, principalmente, os três serviços a seguir:

- **Serviço de Aquisição de Recursos:** obtém, caso existam, os arquivos de entrada que serão necessários para a execução das atividades;
- **Serviço de Execução:** monitora as tarefas, bem como mantém o ciclo de vida das mesmas;
- **Serviço de Persistência:** é chamado ao final de uma tarefa executada com sucesso. Este serviço efetua o *upload*, caso existam, de arquivos produzidos durante a execução da tarefa para a nuvem específica em questão.

## 3.3 Considerações Finais

Neste capítulo foi apresentada a plataforma de federação de nuvens BioNimbuZ 2. Para isso, foram descritos sua arquitetura e seus principais serviços. Como a plataforma de federação ainda não possui integração com nuvens privadas, foi proposto neste trabalho um componente que comunique o BioNimbuZ 2 com um provedor *OpenStack*.

Sob esse contexto, o Capítulo 4 apresentará o *OpenStack*, que é uma das tecnologias essenciais para este projeto, uma vez que esse *software* é um sistema operacional em nuvem que proporciona a criação de nuvens privadas.

# Capítulo 4

# *OpenStack*

As seções seguintes deste capítulo detalham o funcionamento e a arquitetura do *OpenStack* [22], que é um sistema operacional em nuvem escalável e de código aberto. Para isso, a Seção 4.1 apresenta uma visão geral acerca do assunto, enquanto a Seção 4.2 exibe e detalha a arquitetura do sistema. Por fim, a Seção 4.3 trata das considerações finais acerca deste capítulo.

## 4.1 Visão Geral

O *OpenStack* [22] é um sistema operacional de código aberto para criação e gerenciamento de infraestruturas de nuvens, e foi originalmente desenvolvido pela *NASA* (*National Aeronautics and Space Administration*, ou Administração Nacional da Aeronáutica e Espaço), [43] e pelo *Rackspace* [44], que é um provedor de infraestrutura americano. Assim, dentro de um *datacenter*, com ele é possível controlar recursos de computação, de armazenamento e também de rede. Esse sistema permite, portanto, a criação de nuvens tanto públicas quanto privadas.

Por ser um pacote *open-source* e gratuito, permite que pequenos *players* tenham oportunidade de implantar pequenas infraestruturas de nuvem. O *OpenStack* representa atualmente uma das maiores comunidades de desenvolvimento de computação em nuvem de código aberto, e também é o mais amplamente adotado na indústria [45].

Em relação ao seu desenvolvimento, tal plataforma é elaborada e lançada em ciclos de seis meses. Após esse lançamento inicial, as versões anteriores que ainda são mantidas também recebem atualizações. A versão corrente é a *Stain* [46], ao passo que uma nova, chamada de *Train* [47], já está sendo desenvolvida. Atualmente, cinco versões são mantidas (*Stain*, *Rocky*, *Queens*, *Pike* e *Ocata*), todas elas mantendo uma arquitetura bastante similar.

## 4.2 Arquitetura

O *OpenStack* é composto por seis projetos que lidam com serviços básicos e estáveis da computação em nuvem, os quais são computação, rede, armazenamento de blocos, armazenamento de objetos, identidade e imagens. Além desses serviços principais, existem também muitos outros que abordam temas complementares à nuvem, dentre os quais se destaca o *Horizon* [48], que é o serviço de interface gráfica.

Os projetos se comunicam por meio de consistentes APIs (*Application Programming Interfaces*, ou Interfaces de Programação de Aplicação) RESTful que facilitam o uso do sistema operacional, em conjunto com uma boa documentação.

A Figura 4.1 apresenta as principais ferramentas do *OpenStack* e como eles se comunicam entre si. Dentre elas, foram utilizadas diretamente na implementação do componente proposto nesta monografia, o *Nova* [49], o *Swift* [50], o *Glance* [51] e o *Keystone* [52], que serão descritos nas próximas seções.

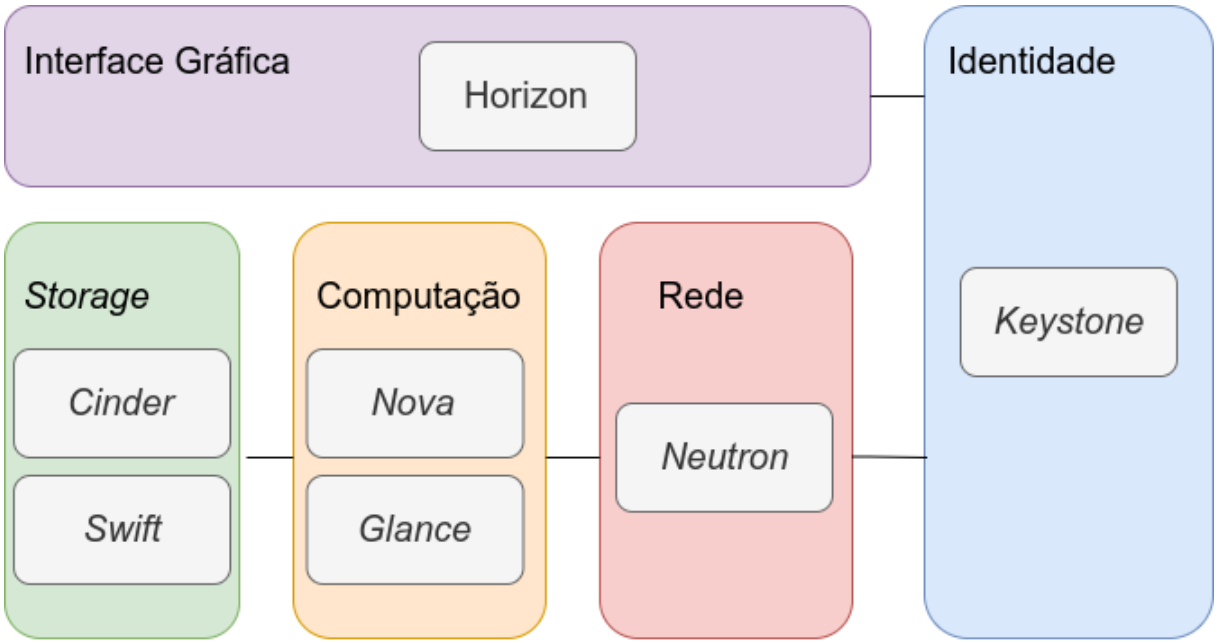


Figura 4.1: Arquitetura das principais ferramentas do *OpenStack*.

### 4.2.1 *Cinder*

A ferramenta *Cinder* é o serviço de *Block Storage* do *OpenStack* e fornece volumes para máquinas virtuais do *Nova*, *hosts* de *hardware* e também *containers*. A implementação deste projeto utilizou, principalmente, a linguagem de programação *Python*. Os principais objetivos deste serviço [53] são:

- **Alta disponibilidade**: altamente escalável e disponível até mesmo para grandes *workloads*;
- **Arquitetura baseada em componentes**: torna-se rápido e fácil adicionar novos comportamentos;
- **Recuperável**: o processo de *debug* é simples e, portanto, diagnosticar e corrigir erros também é simples;
- **Tolerante à falhas**: devido à natureza de processos isolados, um único erro não provoca falhas em cascata.

### 4.2.2 *Swift*

O *Swift* [50] é o projeto que implementa um repositório para armazenamento de objetos. As organizações podem utilizar esse sistema para armazenar uma grande quantidade de dados de maneira eficiente, segura e barata. É altamente tolerante à falhas e também suporta redundância e *streaming*, tanto de áudio quanto de vídeo. É extremamente escalável em termos de tamanho e também de capacidade. Ele foi implementado sob a tecnologia *Python*.

### 4.2.3 *Nova*

O *Nova* [49] é o projeto do OpenStack que fornece maneiras de provisionamento de instâncias de computação, ou seja, serviços virtuais. Esse projeto foi implementado quase inteiramente com a linguagem *Python*, e é executado como um conjunto de *daemons* sobre os servidores *Linux* existentes para fornecer esse serviço.

Para descrever a arquitetura desta ferramenta, destacam-se os seguintes componentes [54]:

- **DB**: banco de dados SQL para armazenamento de dados;
- **API**: componente responsável por receber requisições HTTP, converter comandos e se comunicar com os demais componentes por meio de mensagens ou até mesmo HTTP;
- ***Scheduler***: é o elemento da arquitetura que decide qual *host* se responsabiliza por qual instância;
- ***Compute***: encarregado por gerenciar a comunicação com o *hypervisor* e as máquinas virtuais;

- ***Conductor***: lida com requisições que necessitam de coordenação (compilação/ redimensionamento), atua como um *proxy* de banco de dados e manipula conversões de objetos;
- ***Placement***: rastreia inventários e utilizações de provedores de recursos.

### 4.2.4 *Glance*

A ferramenta *Glance* [51] tem como principal objetivo gerenciar imagens de máquinas virtuais, e para cumprir esse propósito fornece serviços de busca e de armazenamento dessas imagens. Ele foi implementado com a linguagem *Python*.

O *glance* possui uma arquitetura cliente-servidor que fornece uma API REST ao usuário por meio da qual as solicitações ao servidor podem ser executadas. Tal arquitetura possui os seguintes componentes [55]:

- **Cliente**: uma aplicação qualquer que faça uso de um servidor *Glance*;
- **REST API**: é a interface para comunicar o "mundo exterior"com a infraestrutura da ferramenta;
- ***Database Abstraction Layer* (DAL)**: essa camada de abstração é uma API que unifica a comunicação entre o *Glance* e a interface de bancos de dados;
- ***Glance Domain Controller***: é o controlador que implementa as principais funcionalidades deste serviço, como autorização, notificações e conexão com banco de dados;
- ***Glance Store***: é o componente cuja responsabilidade é organizar as interações entre a ferramenta e vários armazenamentos de dados;
- ***Registry Layer***: essa camada é opcional e é usada para organizar uma segura comunicação entre o domínio e o DAL.

### 4.2.5 *Neutron*

O projeto *Neutron* [56] tem como objetivo principal fornecer "*network connectivity as a service*", ou conectividade de rede como um serviço, entre dispositivos de interface gerenciados por outros serviços do *OpenStack*, como por exemplo o *Nova*, explicado na Seção 4.2.3. Tal sistema foi escrito com a tecnologia *Python*. Dentre as vantagens e os pontos positivos do *Neutron*, destacam-se:

- Oferece aos inquilinos (*tenants*) da nuvem uma API para criar topologias de rede complexas e configurar políticas de rede avançadas na nuvem;

- Possibilita uso de *plugins* para introduzir recursos e características específicas para atender necessidades dos projetos;
- Facilidade para criar redes complexas por intermédio do suporte com o projeto *Horizon*, detalhado na Seção 4.2.7.

### 4.2.6 *Keystone*

O *Keystone* [52] é o serviço de identidade do *OpenStack*. É usado para autenticação e autorização dos serviços da plataforma. Além disso, ele também fornece o catálogo de *endpoints* para todos os serviços da plataforma. Esse serviço também foi implementado em *Python*.

O *Keystone* é organizado como um grupo de serviços internos que são expostos por intermédios de diversos *endpoints* que são disponibilizados pelas APIs. Esses serviços são:

- ***Identity***: esse serviço fornece validação das credenciais de autenticação e informações acerca de usuários e grupos de usuários;
- ***Resource***: fornece informações sobre *domains*, ou domínios, e sobre *projects*, ou projetos;
- ***Assignment***: provê informações a respeito de *roles* (que ditam nível de autorização) e também *role assignments* (uma 3-upla que possui um *role*, um *resource* e uma *identity*);
- ***Token***: é o serviço que valida e gerencia os *tokens* usados para autenticação uma vez que uma credencial de um usuário já foi verificada;
- ***Catalog***: esse componente do projeto fornece um catálogo dos *endpoints* principais do sistema;
- ***Policy***: provê um mecanismo de autorização baseado em regras.

### 4.2.7 *Horizon*

O *Horizon* [48] é um painel de controle para que os usuários e os administradores possam interagir com a plataforma, e também possam usar os serviços da nuvem por intermédio de uma interface gráfica. Para a implementação deste projeto, as principais linguagens de programação utilizadas foram o *Python* e o *JavaScript*.

O *Horizon* possui alguns valores no núcleo de seu *design* e de sua arquitetura. São eles:

- **Suporte**: possui suporte aos principais projetos *OpenStack*;

- **Extensível**: como sendo código aberto, está livre para desenvolvedores adicionarem novas funcionalidades;
- **Gerenciável**: a base do código principal deve ser simples e de fácil navegação;
- **Consistente**: modelos visuais e interações são consistentes e mantidos por toda a plataforma;
- **Estável**: possui uma API confiável com ênfase e destaque na compatibilidade com versões anteriores;
- **Usável**: interface simples e eficiente para que usuários consigam tenham experiências agradáveis ao utilizar o painel de controle.

### 4.2.8 Outros Projetos

Além dos componentes do *OpenStack* já citados e elucidados, existem também muitos outros projetos de suporte à nuvem com diferentes propósitos, como por exemplo orquestração, disponibilidade, otimização e coleta de dados. Dentre esses componentes, destacam-se os seguintes:

- ***Heat***: ferramenta para orquestrar os recursos de infraestrutura para um aplicativo em nuvem [57]. Além disso, o *Heat* também disponibiliza um serviço de escalonamento automático;
- ***Masakari***: o propósito fundamental desse componente é garantir uma alta disponibilidade das nuvens *OpenStack* ao recuperá-las quando ocorrer alguma falha [58];
- ***Ceilometer***: esse projeto coleta, normaliza e transforma de uma maneira eficiente os dados produzidos pelos serviços de um sistema *OpenStack*. Com tais dados, o *Ceilometer* gera diversas visualizações para o usuário;
- ***Watcher***: esta ferramenta foca nas nuvens *OpenStack* com vários inquilinos, ou "*multi-tenant*". O serviço oferecido de otimização dos recursos é flexível e escalonável.

## 4.3 Considerações Finais

Neste capítulo foi apresentado em detalhes o sistema OpenStack, bem como sua arquitetura e suas principais ferramentas. Tal sistema não possui integração com a plataforma de federação de nuvens BioNimbuZ 2.

Assim, o próximo capítulo apresenta a implementação do componente que fará essa integração, e permitirá que o BioNimbuZ 2 seja capaz de se comunicar com nuvens privadas do *OpenStack*.

# Capítulo 5

# *Componente OpenStack*

Neste capítulo são revelados os detalhes do componente proposto nesta monografia. Na Seção 5.1 estão os detalhes acerca das tecnologias e das metodologias utilizadas. A Seção 5.2 examina os métodos e os testes necessários para garantirem o funcionamento do componente. A Seção 5.3 mostra os resultados das execuções de *workflows*. Por fim, na Seção 5.4, são apresentadas as considerações finais sobre a implementação feita.

## 5.1 Visão Geral

Tendo em vista a arquitetura do BioNimbuZ 2, o componente proposto é um microsserviço e, portanto, é totalmente independente do funcionamento global da plataforma. Ele será incorporado na Camada de Federação do BioNimbuZ 2. A Figura 5.1 mostra como os componentes estão relacionados com a Camada de Aplicação, no qual tal camada envia pacotes *json* que são lidos pelos componentes, que interpretam as informações e as transmitem à sua respectiva nuvem. O objetivo principal deste novo artefato proposto nesta monografia é garantir a comunicação entre a plataforma de federação de nuvens e um sistema *OpenStack*.

Tendo em vista que tal comunicação é padronizada entre os provedores e que a arquitetura da plataforma de federação de nuvens é orientada à microsserviços, a atualização e a execução do componente não provoca interrupções na Camada de Aplicação, nem em qualquer outra camada do sistema.

Para o desenvolvimento do componente, a linguagem de programação escolhida foi o Java [59], com auxílio da tecnologia *Spring Boot* [60]. Dentre os principais motivos para essas escolhas, estão:

- **Padronização**: são as mesmas tecnologias utilizadas para a elaboração dos demais componentes, portanto tais parâmetros foram mantido;

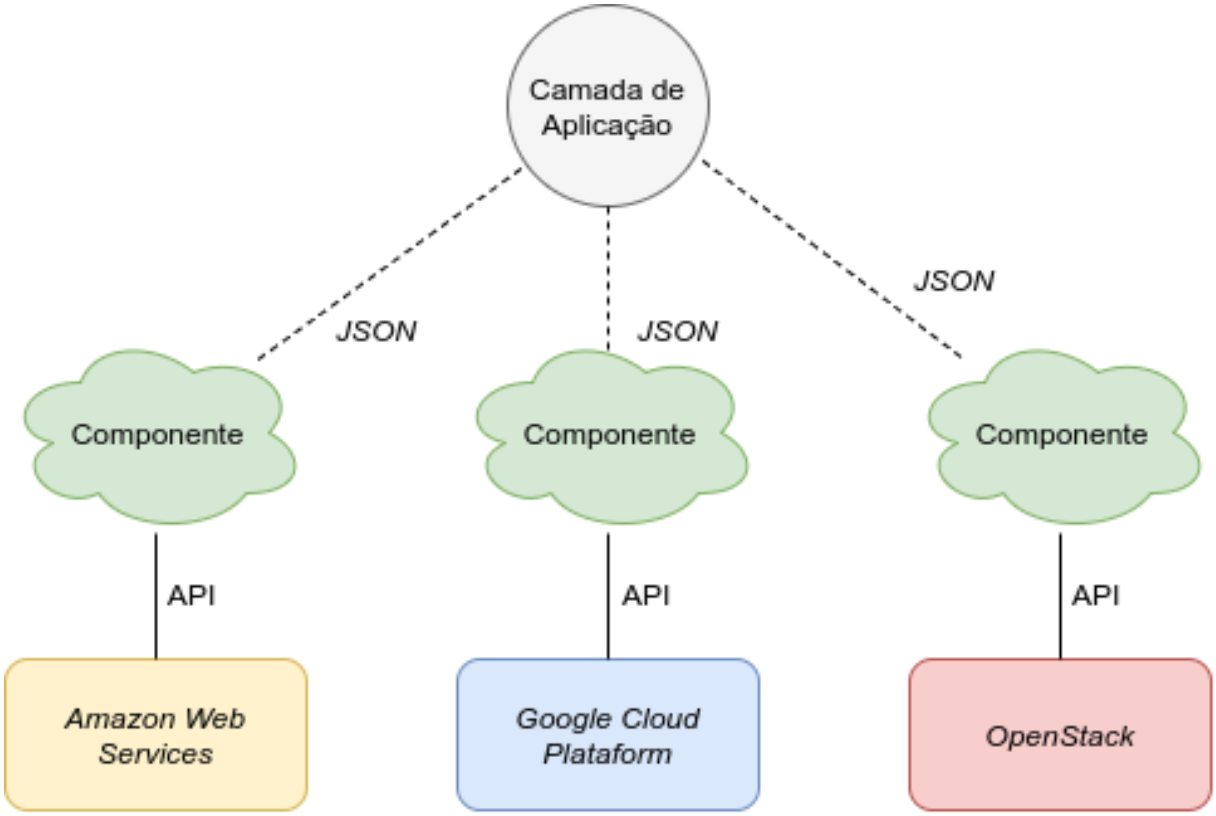


Figura 5.1: Estrutura de comunicação entre Camada de Aplicação e componentes.

- **Facilidade para contribuição**: o Java é uma linguagem amplamente conhecida e utilizada [61], desta forma é mais fácil que outros programadores contribuam com o projeto, uma vez que o BioNimbuZ 2 é uma iniciativa de código aberto;
- **Ferramentas disponíveis**: existem SDKs (*Software Development Kit*, ou *kit* de desenvolvimento de *software*) que implementam de forma fluente as APIs do *OpenStack* para Java, como será explicado um pouco adiante. Esses *kits* facilitam o processo de desenvolvimento.

O *OpenStack* fornece APIs para todas as suas ferramentas, a fim de que, em conjunto, sejam aptas a administrar todos os recursos disponíveis desse sistema. Ao se aproveitar disso, existem projetos que se utilizam de tais APIs para criar SDKs para a linguagem Java, dentre os quais se destacam o *OpenStack4j* [62] e o *jclouds* [63]. O escolhido para ser utilizado neste projeto foi o *OpenStack4j*. As principais motivações de tal escolha foram a facilidade do seu uso e a boa documentação.

Para o controle e o versionamento foi utilizado o *Git*, com o *GitHub* [64] e o repositório é público e está disponível online em *https://github.com/bionimbuz/BionimbuzWeb*.

Assim, foi criado um *branch* a partir do ramo principal do projeto BioNimbuZ 2 para o desenvolvimento do componente proposto neste trabalho.

Em relação ao estilo de desenvolvimento, o TDD (*Test Driven Development*, ou Desenvolvimento Dirigido à Testes) foi adotado [65]. Nele, o código de produção é escrito após os testes do mesmo. Ou seja, antes de criar uma funcionalidade nova, os seus testes devem ser avaliados e criados.

Assim, durante a fase de desenvolvimento do componente, com finalidade de simplificar a infraestrutura envolvida na elaboração do projeto, o sistema *OpenStack* foi instalado em uma máquina virtual no *Google Compute Engine* [21], onde estava inicializado o sistema operacional *CentOS* 7 [66].

A instalação citada acima do *OpenStack* ocorreu por intermédio do *RDO Packstack* [67], que é um *software* que utiliza módulos *Puppet* para implantar vários componentes do *OpenStack* em diversos servidores pré-instalados por meio de SSH automaticamente.

## 5.2 Implementação

A implementação do componente proposto nesta monografia pode ser decomposto em três divisões princiais: configuração, código de produção e teste. A Figura 5.2 demonstra como o componente se comporta internamente e também como ele se encaixa no projeto. Em seguida, essas três divisões da implementação serão detalhadas.

### 5.2.1 Configuração

Nesta parte do projeto, foram criados também três importantes arquivos:

- ***/plugin-openstack/config/application.properties***: por padrão, a porta utilizada pelo *SpringBoot* para o servidor da aplicação é 8080. Essa porta foi alterada para a 8686, com a finalidade de funcionar em uma porta diferente dos demais componentes do projeto BioNimbuZ 2;
- ***/plugin-openstack/.gitignore***: os caminhos e os arquivos descritos no *gititnore* não são rastreados pelo Git. Arquivos específicos, locais e exclusivos de desenvolvimento foram adicionados neste arquivo, uma vez que não há necessidade de versionar tais arquivos;
- ***/plugin-openstack/pom.xml***: o *POM*, *Project Object Model*, é um dos principais arquivos de um projeto *Maven*. Ele possui informações de configurações de dependências do projeto, e é onde foi adicionado o repositório do *OpenStack4j*, para que o próprio projeto se encarregue de buscar tal repositório e torná-lo acessível em todo o projeto.

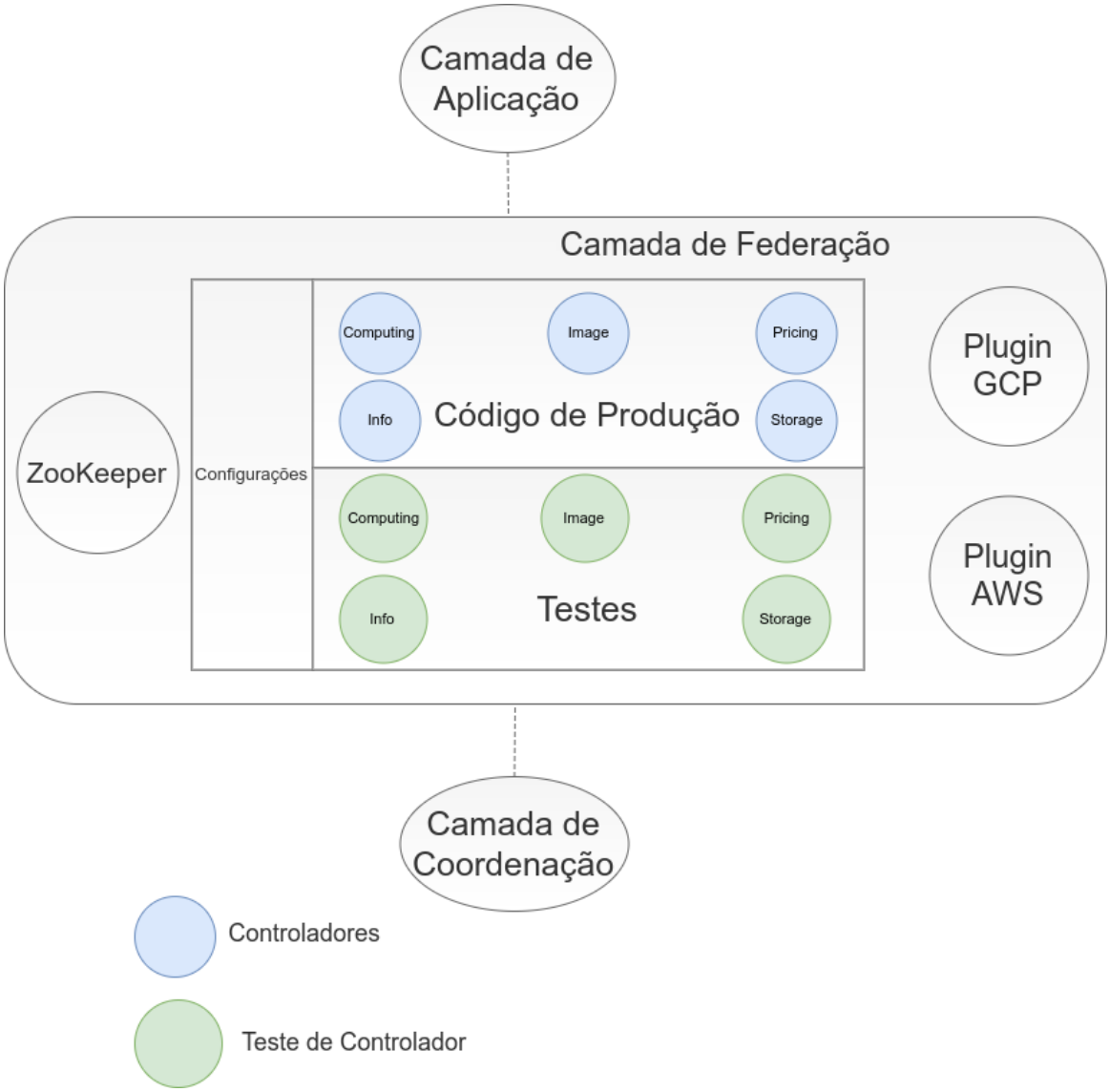


Figura 5.2: Relacionamento entre o componente *OpenStack* e o restante do sistema Bio-NimbuZ 2.

### 5.2.2 Código de Produção

Nesta etapa, está a principal parte da implementação do componente, pois é onde as ações efetivamente ocorrem. Está presente na pasta a "*/plugin-openstack/src/main/java/app*", onde existem ainda dois diretórios, *common* e *controllers*. A pasta *common* possui três arquivos de suporte:

- ***OSClientHelper***: possui um dos métodos fundamentais do projeto, que é o método que retorna o objeto *OSClient*, da blibliteca do *OpenStack4j*. Esse objeto permite acessar as APIs do *OpenStack* diretamente;

- ***PriceTableSingleton***: considerando que o *OpenStack* utiliza os recursos do próprio usuário, o conceito de "tabela de preços"não faz sentido neste componente. Sendo assim, o *PriceTableSingleton* é um *Singleton* [68] que cria um objeto cujos preços relacionados às instâncias e aos volumes é sempre zero;
- ***SystemConstants***: constantes gerais do sistema, como "*PLUGIN_VERSION*"e "*PLUGIN_NAME*".

Já no diretório *controllers*, existem cinco controladores que implementam uma classe abstrata previamente definida e, portanto, sobrescrevem os métodos anteriormente escritos. A finalidade disso é padronizar os componentes do projeto. Os controladores são os seguintes:

- ***ComputingController***: trata diretamente com o gerenciamento das instâncias: criação, captação, exclusão e listagem dessas instâncias. Além disso, também lista as regiões e as zonas das regiões disponíveis. O componente *OpenStack Nova* [49] foi necessário para a elaboração deste *controller*;
- ***ImageController***: lida com as imagens do projeto, ao listar e recuperar as mesmas do sistema. O controlador utiliza o serviço *Glance* [51];
- ***InfoController***: responsável por fornecer informações genéricas acerca do componente, como o tipo de nuvem e a versão do sistema;
- ***PricingController***: trata da precificação das máquinas virtuais e dos volumes;
- ***StorageController***: cria e deleta *containers* de armazenamento de objetos. Utiliza o projeto *Swift* [50].

O componente *Keystone* [52], por lidar com o acesso à API do *OpenStack*, também foi necessário em todos os controladores citados.

A maior parte dos métodos implementados possui uma estrutura parecida, descrita a seguir:

1. **Adquirir *Client* do *OpenStack4j***: a partir das credencias cadastradas pelo usuário corrente e do *token* gerado, é criado um cliente do *OpenStack4j* para ter um fácil acesso às APIS do *OpenStack*;
2. **Comunicar com API OpenStack**: de acordo com cada método, é feita a chamada à API;
3. **Retornar valores**: o valor retornado no passo anterior é tratado para ajustar-se de acordo com os modelos do projeto e do banco de dados;

4. ***Logging* de erros**: caso tenha ocorrido algum erro durante a execução método, o sistema imprime um *log* de acordo com o erro acontecido.

A Figura 5.3 exemplifica esses passos com o método *listIntances* e os números em vermelho na imagem mostram os passos citados.

```
@Override
protected ResponseEntity<Body<List<PluginComputingInstanceModel>>> listInstances(final String token,
        final String identity) throws Exception {
    try {

        Map<String,String> clientData = retrieveCredentialData(identity);
    1: OSClient.OSClientV3 os = getOSClient(token, clientData.get("host"), clientData.get("project_id"));

    2: List<? extends Server> servers = os.compute().servers().list();
    3: return ResponseEntity.ok(Body.create(serversToPluginModel(servers)));
    } catch (final Exception e) {
    4: LOGGER.error(e.getMessage(), e);
    }
    return null;
}
```

Figura 5.3: Método *listIntances*.

### 5.2.3 Teste

Como já foi descrito antes, o primeiro passo para o desenvolvimento de um novo recurso é a elaboração de seu teste. Portanto, é imporante que eles sejam bem escritos e, em conjunto com testes manuais da aplicação, garantissem o funcionamento do projeto.

Os testes estão presentes na pasta "*/plugin-openstack/src/test/java*". Como para acessar o sistema é necessário uma credencial válida com dados sobre um sistema existente *OpenStack*, ao executar os testes, é necessário enviar como parâmetro o endereço de um arquivo *json* que contenha as seguintes chaves válidas: *user*, *password*, *project_id* e *host*. É o mesmo arquivo utilizado no sistema BioNimbuZ 2 para cadastro da credencial *OpenStack*.

Dentro do diretório citado, existem outros dois diretórios, *utils* e *app.controller*. O primeiro citado possui arquivos para auxiliar a leitura do arquivo json referenciado na linha de comando, e também cria uma entidade HTTP básica utilizada por todos os testes que fazem chamadas REST para o sistema.

O outro diretório, *app.controller*, possui de fato os testes, que são relacionados aos controladores. Todos os métodos de tais controladores foram testados. A tecnologia utilizada para a confecção dos teste foi o *JUnit* [69]. Os arquivos desse diretório são: *ComputingControllerTest*, *ImageControllerTest*, *InfoControllerTest*, *PricingControllerTest* e *StorageControllerTest*. A maioria dos testes possui a seguinte estrutura:

1. **Recuperar credencial**: leitura e tratamento do arquivo da credencial fornecida pelo o executor do teste;
2. **Gerar *token***: gera-se um *token* de autenticação do *OpenStack* a partir dos dados da credencial obtida;
3. **Retornar entidade HTTP:** cria-se um *template* de uma requisição HTTP;
4. **Efetuar requisição:** é feita a requisição (ou *request*) de acordo com o controlador e o método a ser testado;
5. **Conferir valores retornados**: a resposta (ou *response*) é analisada e comparada com a expectativa.

A Figura 5.4 exemplifica os três primeiros passos, com o método *createEntity*. A Figura 5.5 mostra mostra o método *getImageTest* com os dois últimos passos. Os números em vermelhor mostram onde cada passo está sendo executado.

```
public static <T> HttpEntity<T> createEntity(T content) {
    OSFactory.enableHttpLoggingFilter( enabled: true);
1:  String credential = readCredential();

    JsonParser parser = new JsonParser();
    JsonObject json = (JsonObject) parser.parse(credential);
    String identity = json.get("host").getAsString() + "/" + json.get("project_id").getAsString();


    try {
2:      TokenModel tokenModel = Authorization.getToken(SystemConstants.CLOUD_TYPE,  scope: null, credential);

        Map<String,String> clientData = retrieveCredentialData(identity);

        OSClient.OSClientV3 os = getOSClient(tokenModel.getToken(), clientData.get("host"), clientData.get("project_id"));

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        headers.add(HttpHeadersCustom.AUTHORIZATION_ID, identity);
        headers.add(HttpHeaders.AUTHORIZATION, os.getToken().getId());
        headers.add(HttpHeadersCustom.API_VERSION, GlobalConstants.API_VERSION);

        HttpEntity<T> entity =
                new HttpEntity<>(content, headers);

3:      return entity;
    } catch (Exception e) {
        e.printStackTrace();
        assertThat(e).isNull();
        return null;
    }
}
```

Figura 5.4: Método *createEntity*.

## 5.3 Resultados

Para demonstrar os resultados do componente, primeiro foi instalado o *OpenStack* sob o sistema operacional CentOS no LaBiD, o Laboratório de Bioinformática e Dados, utili-

```
public void getImageTest() {
    TestUtils.setTimeout(restTemplate.getRestTemplate(),  timeout: 0);
3:  HttpEntity<Void> entity = TestUtils.createEntity( content: null);

4:  ResponseEntity<Body<PluginImageModel>> response =
            this.restTemplate
                .exchange(
                        url: Routes.IMAGES +"/"+ imageId,
                        HttpMethod.GET,
                        entity,
                        new ParameterizedTypeReference<Body<PluginImageModel>>() {});
5:  assertThat(response).isNotNull();
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
}
```

Figura 5.5: Método *getImageTest*.

zando a instalação *RDO Packstack* [67]. Assim, foi realizada uma sequência de tarefas para demonstrar em execução o que foi implementado.

Antes de inicializar o sistema BioNimbuZ 2, é necessário efetuar alguns ajustes no *OpenStack*. Primeiro, foram criadas duas redes, uma pública, com configurações da rede do LaBiD, e outra privada, com configurações da rede interna do OpenStack. Em seguida, criou-se um roteador que efetuasse a comunicação entre as duas redes. A topologia final está apresentada na Figura 5.6.

Assim, é possível iniciar o BioNimbuZ 2. O primeiro passo do usuário é a autenticação no projeto. Ele deve enviar ao sistema uma arquivo *json* que contenha as seguintes chaves: *user*, *password*, *project_id*, *host*. A Figura 5.7 ilustra um exemplo desse arquivo *json*. Para enviar tal arquivo, deve-se acessar a opção *Credentials* do menu principal, e depois *Add Credential*. Nessa página, como mostrado na Figura 5.8, escolhe-se um nome para a credencial e adiciona-se o arquivo na opção *Browse*.

O segundo passo para o usuário é adicionar uma imagem disponível no *OpenStack*. No menu principal, deve ir em *Images* e então *Add Image*. Na tela apresentada na Figura 5.9, seleciona-se *OpenStack* no *dropdown Plugin* e escolhe-se a imagem que preferir.

O terceiro passo é adicionar uma aplicação para ser executada na instância. Para este projeto, usou-se a indicação de Angelo *et. al* [70] para o *workflow* de Astronomia, no qual o usuário acessa a opção *Executor*, sob o menu *Applications* e insere o *script* da tarefa no campo *Execution Script*. Ao finalizar essa etapa, o usuário seleciona a aplicação adicionada e, como exposto na Figura 5.10, escolhe a nova imagem para que fique disponível a executar a aplicação.

No quarto passo, ainda seguindo o que foi exposto por Angelo *et. al* [70], o usuário cria o *workflow*, no menu *Executions*, depois na opção *Workflows*. O quinto passo trata da criação da instância e da execução do *workflow*. O usuário acessa *Instances*, dentro

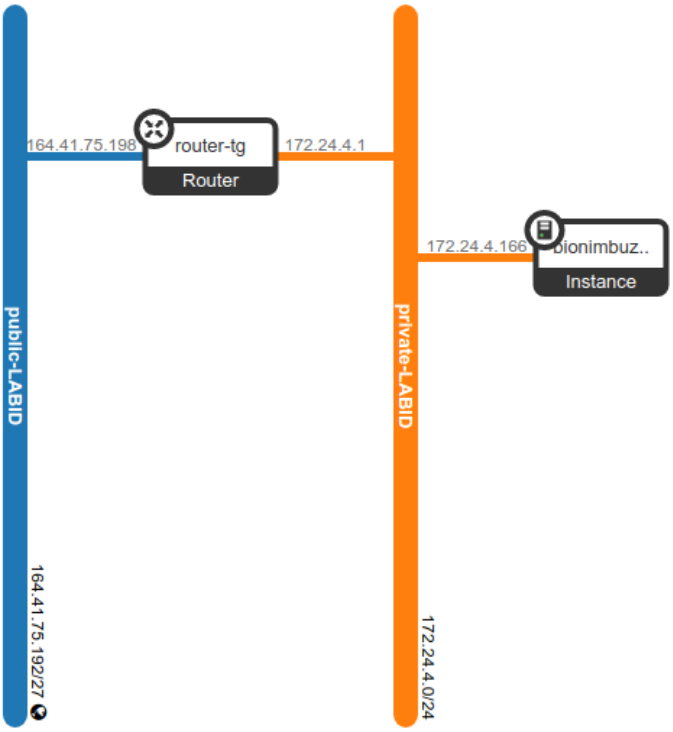


Figura 5.6: Topologia de rede do *OpenStack*.

```
{
  "user": "admin",
  "password": "PASSWORD",
  "project_id": "PROJECT_ID",
  "host": "164.41.75.202"
}
```

Figura 5.7: Exemplo de arquivo de cadastro.

do menu *Executions* e então *Add Instance*. Ao selecionar em *Executor* a aplicação adicionada no terceiro passo, deve marcar o *checkbox Request execution after creation*. O formulário é concluído de maneira intuitiva, como mostrado na Figura 5.11. Ao salvar, a instância é criada, mas, apesar dos esforços, não foi possível efetuar as configurações corretas envolvendo a rede do LaBiD e o *OpenStack* instalado nele, de maneira que não foi possível acessar as instâncias criadas pelo *OpenStack* externamente. Assim, não foi possível executar o *script*.

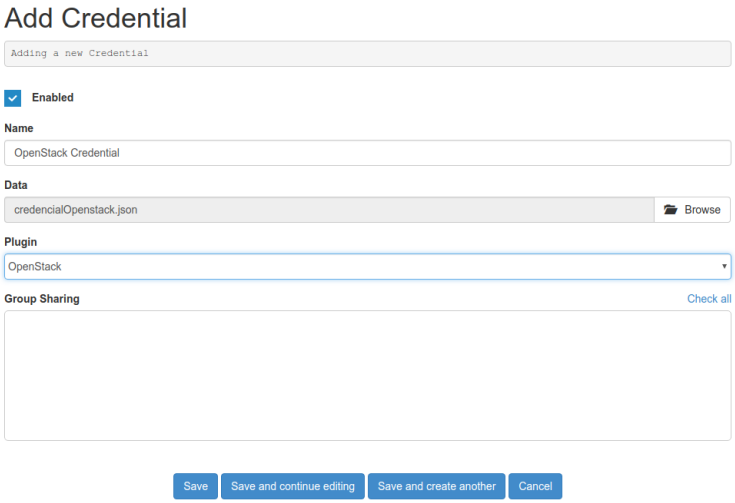


Figura 5.8: Tela para adicionar credencial.

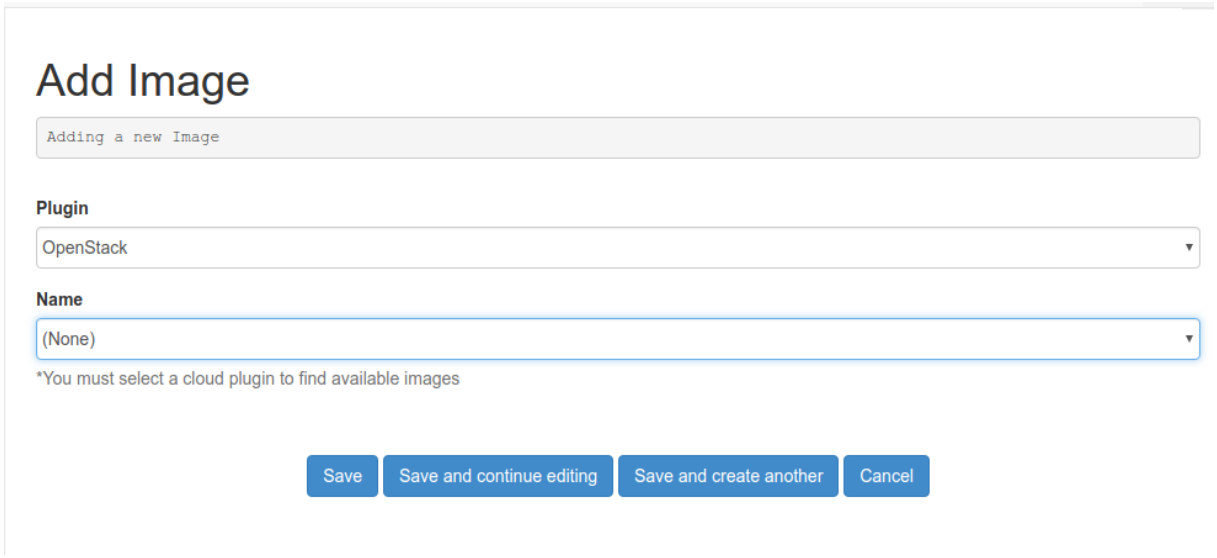


Figura 5.9: Tela para adicionar imagem.

## 5.4 Considerações Finais

Neste capítulo foram apresentados os detalhes acerca da implementação no que tange à tecnologias e a bibliotecas utilizadas, técnicas de desenvolvimento, versionamento e nuances dos controladores. Além disso, foi mostrado também o componente sendo posto em prática e executando os controladores e seus métodos que foram detalhados neste capítulo, apesar de que o *script* final do *workflow* não foi executado integralmente. Ainda assim, ficou demonstrado que é possível integrar nuvens privadas ao projeto BioNimbuZ

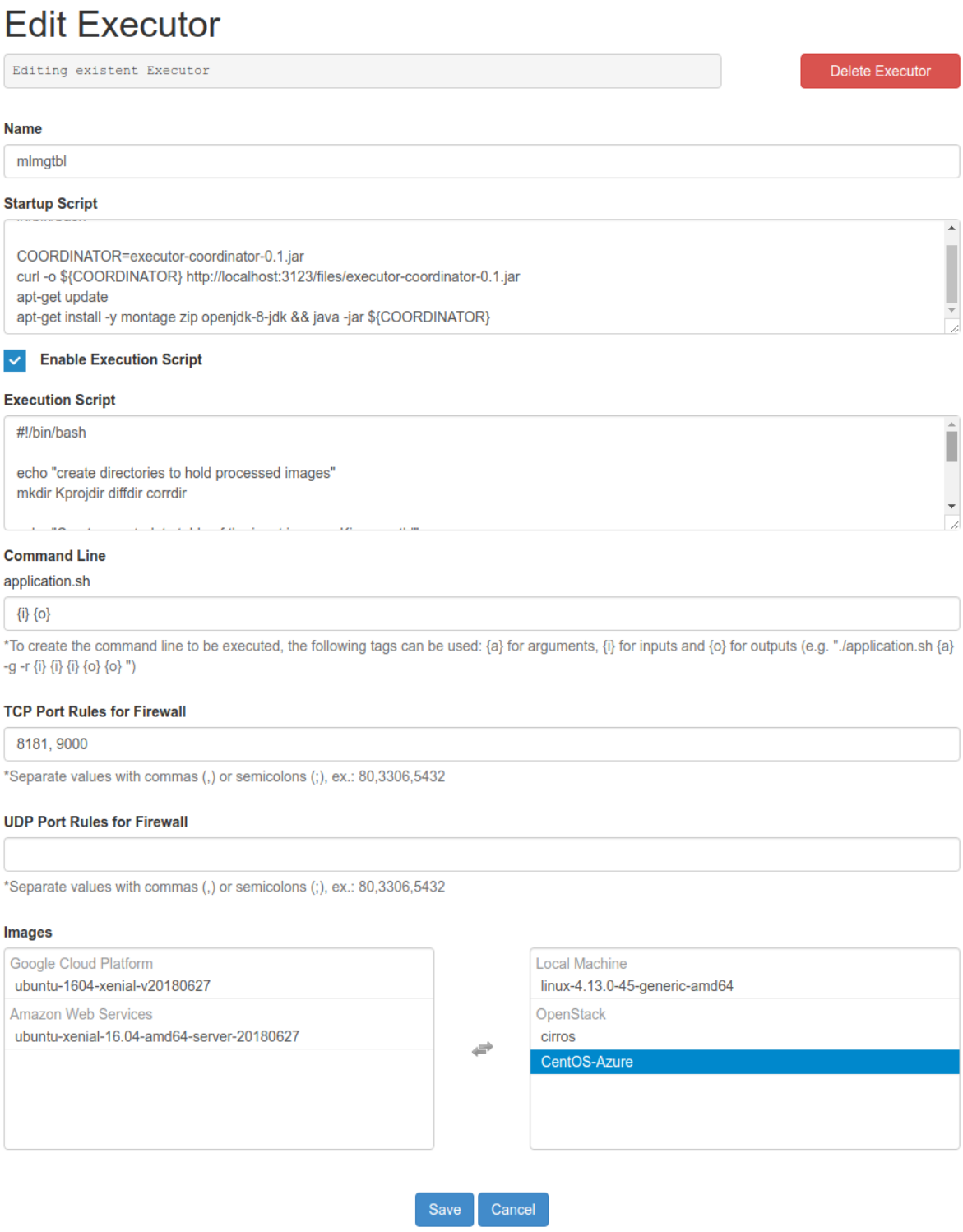


Figura 5.10: Tela para editar *executor*.

2. Para finalizar esta monografia, o Capítulo 6 apresenta as conclusões deste trabalho e

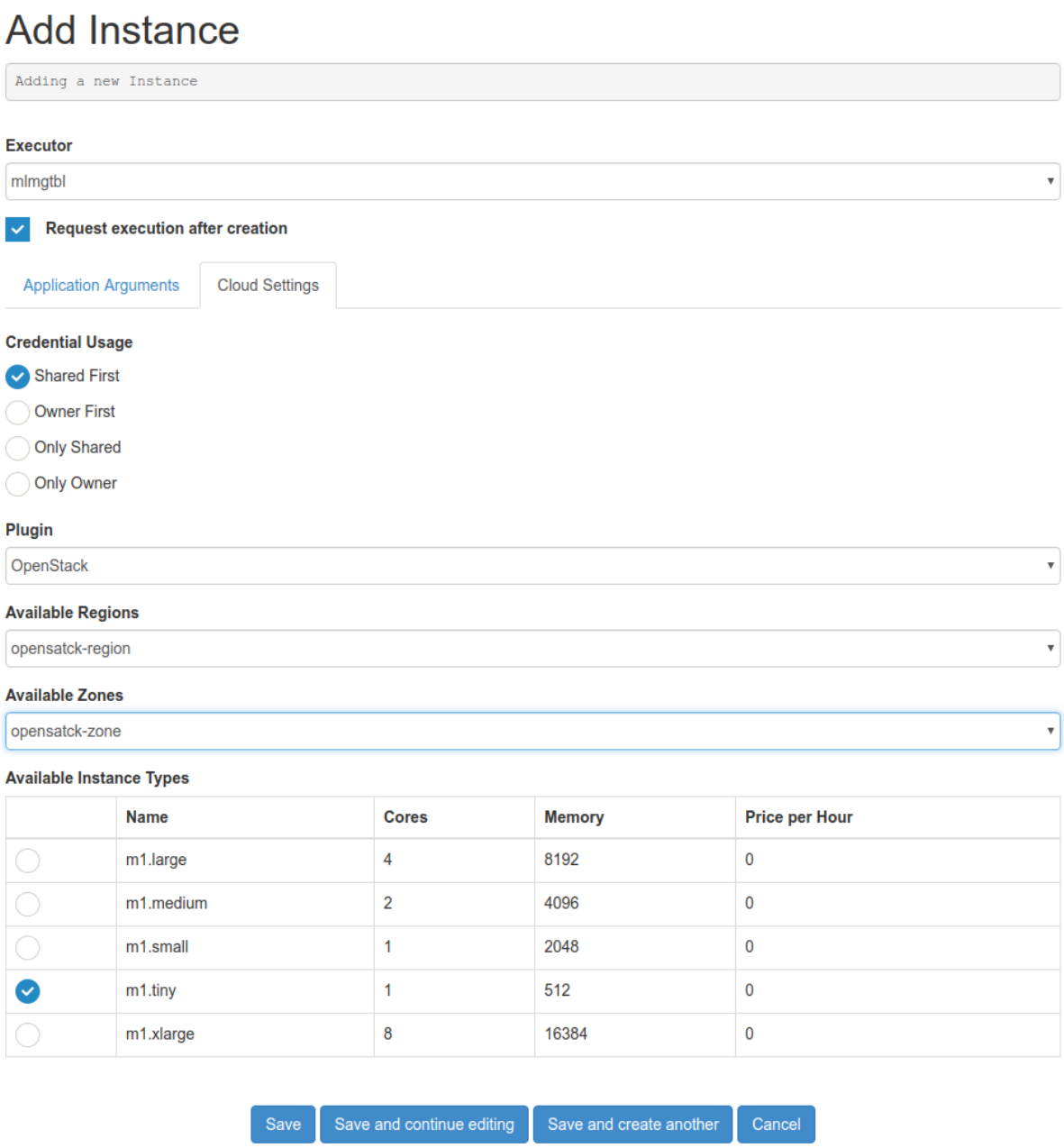


Figura 5.11: Tela para adicionar instância.

os trabalhos futuros considerados.

# Capítulo 6

# Conclusão

Sob o contexto de um cenário de crescente demanda computacional pela indústria e pela academia, foi desenvolvido o BioNimbuZ 2 [1], que é uma plataforma de federação de nuvens orientada a microsserviços capaz de executar *workflows* de diversas áreas científicas em diferentes provedores de nuvens.

Por outro lado, o BioNimbuZ 2 não apresentava suporte para nuvens privadas. Assim, foi implementado o componente *OpenStack* proposto nesta monografia, que supre essa necessidade e abre possibilidade também para execução de *workflows* de maneira híbrida, alternando a execução das tarefas em nuvens públicas e privadas.

Isso faz com que a plataforma BioNimbuZ 2 eleve seu patamar ao deixar de ser uma plataforma de federação de nuvens públicas para uma plataforma de federação de nuvens híbridas.

Todavia, como apresentado neste trabalho, existem vários desafios inerentes à implementação de uma plataforma de federações de nuvens com a utilização de microsserviços. Dentre eles, estão a falta de padrões abertos entre os diferentes sistemas que são integrados ao BioNimbuZ 2, a utilização de sistemas distribuídos e também problemas relacionados à segurança e ao monitoramento dos recursos.

Para os trabalhos futuros, devem ser consideradas alternativas para solucionar os problemas que limitaram este trabalho. Entre elas, está a implementação, no BioNimbuZ 2, de um novo componente para outra plataforma de nuvens privadas, tal como o *CloudStack*. Também outra alternativa é buscar um ambiente *OpenStack* que esteja plenas condições para a execução os testes.

Além disso, deve ser considerado um trabalho de comparação entre o uso de diferentes provedores de nuvens no BioNimbuZ 2 ao executar diferentes *workflows*, levando também em conisderação o uso de nuvens privadas e públicas tanto de maneira excluvisa quanto de maneira híbrida.

Em relação ao BioNimbuZ de maneira genérica, pode-se have um trabalho que trate do monitoramento dos microsserviços. Além disso, promover a integração entre o BioNimbuZ e sistemas como Docker [71] e Kubernets [72].

Para finalizar, deve-se também ponderar alternativas para otimizar e facilitar a arquitetura orientada à microsserviços, como a utilização de ambientes de desenvolvimentos menos ligados ao *back-end* por intermédio de diversos *frameworks* e bibliotecas *JavaScript* disponíveis, tais como Angular [73], Ember [74] e Vue [75].

# Referências


[1] Mendes, Felipe Lopes de Souza: *Bionimbuz 2 - uma plataforma de federação de nuvens em uma arquitetura orientada a microsserviços*. 2018. ix, 2, 12, 13, 38

[2] Vaquero, Luis M, Luis Rodero-Merino, Juan Caceres e Maik Lindner: *A break in the clouds: towards a cloud definition*. ACM SIGCOMM Computer Communication Review, 39(1):50–55, 2008. 1, 5

[3] Mell, Peter, Tim Grance *et al.*: *The nist definition of cloud computing*. 2011. 1, 4, 6

[4] Saldanha, Hugo, Edward Ribeiro, Carlos Borges, Aletéia Araújo, Ricardo Gallon, Maristela Holanda, Maria Emilia Walter, Roberto Togawa e Joao Carlos Setubal: *Towards a hybrid federated cloud platform to efficiently execute bioinformatics workflows*. Em *BioInformatics*. IntechOpen, 2012. 1, 9

[5] Buyya, Rajkumar, Rajiv Ranjan e Rodrigo N Calheiros: *Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services*. Em *International Conference on Algorithms and Architectures for Parallel Processing*, páginas 13–31. Springer, 2010. 1

[6] Celesti, Antonio, Francesco Tusa, Massimo Villari e Antonio Puliafito: *How to enhance cloud architectures to enable cross-federation*. Em *2010 IEEE 3rd international conference on cloud computing*, páginas 337–345. IEEE, 2010. 1, 7

[7] Oliveira, Daniel de, Eduardo Ogasawara, Fernanda Baião e Marta Mattoso: *Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows*. Em *2010 IEEE 3rd International Conference on Cloud Computing*, páginas 378–385. IEEE, 2010. 1

[8] Demchenko, Yuri, Canh Ngo, Cees de Laat e Craig Lee: *Federated access control in heterogeneous intercloud environment: Basic models and architecture patterns*. Em *2014 IEEE International Conference on Cloud Engineering*, páginas 439–445. IEEE, 2014. 1

[9] Kim, Hyunjoo e Manish Parashar: *Cometcloud: An autonomic cloud engine*. Cloud Computing: Principles and Paradigms, páginas 275–297, 2011. 1

[10] Rosa, Michel, Breno Moura, Guilherme Vergara, Lucas Santos, Edward Ribeiro, Maristela Holanda, Maria Emília Walter e Aletéia Araújo: *Bionimbuz: A federated cloud platform for bioinformatics applications*. Em *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, páginas 548–555. IEEE, 2016. 1

[11] Azevedo, Diego Rodrigues e Tarcísio Batista de Freitas Júnior: *Biocirrus: uma nova política de armazenamento para a plataforma bionimbuz de nuvem federada*. 2016. 2

[12] Costa, Heitor Henrique de Paula Moraes: *Controle de acesso na plataforma de nuvem federada bionimbuz*. 2015. 2

[13] Júnior, Barreiros e Willian de Oliveira: *Escalonador de tarefas para o plataforma de nuvens federadas bionimbuz usando beam search iterativo multiobjetivo*. 2016. 2

[14] Moura, Breno Rodrigues de: *Arquitetura de um controlador de sla para ambiente de nuvens federadas*. 2017. 2

[15] Ramos, Vinícius De Almeida: *Um sistema gerenciador de workflows científicos para a plataforma de nuvens federadas bionimbuz*. 2016. 2

[16] Saldanha, Hugo Vasconcelos: *Bionimbus: uma arquitetura de federação de nuvens computacionais híbrida para a execução de workflows de bioinformática*. 2012. 2

[17] Vergara, Guilherme Fay: *Arquitetura de um controlador de elasticidade para nuvens federadas*. 2017. 2

[18] Mattoso, Marta, Jonas Dias, Flavio Costa, Daniel de Oliveira e Eduardo Ogasawara: *Experiences in using provenance to optimize the parallel execution of scientific workflows steered by users*. Em *Workshop of Provenance Analytics*, volume 1, 2014. 2

[19] Raicu, Ioan, Ian T Foster e Yong Zhao: *Many-task computing for grids and supercomputers*. Em *2008 workshop on many-task computing on grids and supercomputers*, páginas 1–11. IEEE, 2008. 2, 15

[20] Amazon: *A. w. s. amazon elastic compute cloud (ec2)*, 2019. `https://aws.amazon.com/pt/`, Acessado em 17 de fevereiro de 2019. 2, 7

[21] Google: *Google engine: Iaas | compute engine | google cloud*, 2019. `https://cloud.google.com/compute`, Acessado em 17 de fevereiro de 2019. 2, 7, 28

[22] OpenStack: *Build the future of open infrastructure*, 2019. `https://www.openstack.org/`, Acessado em 14/03/2019. 2, 19

[23] Buyya, Rajkumar, Chee Shin Yeo, Srikumar Venugopal, James Broberg e Ivona Brandic: *Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility*. Future Generation computer systems, 25(6):599–616, 2009. 4

[24] JoSEP, Anthony D, RAnDy KAtz, AnDy KonWinSKi, LEE Gunho, DAViD PAttERSon e ARiEL RABKin: *A view of cloud computing*. Communications of the ACM, 53(4), 2010. 4, 7

[25] Badger, Lee, Tim Grance, Robert Patt-Corner, Jeff Voas *et al.*: *Cloud computing synopsis and recommendations*. NIST special publication, 800:146, 2012. 5

[26] Toosi, Adel Nadjaran, Rodrigo N Calheiros e Rajkumar Buyya: *Interconnected cloud computing environments: Challenges, taxonomy, and survey.* ACM Computing Surveys (CSUR), 47(1):7, 2014. 7, 8

[27] Bittman, Thomas: *The evolution of the cloud computing market. gartner blog network*, 2008. 7

[28] Microsoft: *Microsoft azure*, 2019. `https://azure.microsoft.com/pt-br/`, Acessado em 17 de fevereiro de 2019. 7

[29] Goiri, Inigo, Jordi Guitart e Jordi Torres: *Characterizing cloud federation for enhancing providers' profit.* Em *2010 IEEE 3rd International Conference on Cloud Computing*, páginas 123–130. IEEE, 2010. 8

[30] Barril, Jose Farnesio Huesca, Jeff Ruyter e Qing Tan: *A view on internet of things driving cloud federation.* Em *2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, páginas 221–226. IEEE, 2016. 8

[31] Kurze, Tobias, Markus Klems, David Bermbach, Alexander Lenk, Stefan Tai e Marcel Kunze: *Cloud federation.* Cloud Computing, 2011:32–38, 2011. 8

[32] Satzger, Benjamin, Waldemar Hummer, Christian Inzinger, Philipp Leitner e Schahram Dustdar: *Winds of change: From vendor lock-in to the meta cloud.* IEEE internet computing, 17(1):69–73, 2013. 9

[33] Fowler, Martin e Lewis, James: *Microservices*, 2019. `https://martinfowler.com/articles/microservices.html`, Acessado em 21 de fevereiro de 2019. 9

[34] Liang Guo, Frank Lin e Junjie Guan: *Building services at airbnb, part 1*, 2019. `https://medium.com/airbnb-engineering/building-services-at-airbnb-part-1-c4c1d8fa811b`, Acessado em 21 de fevereiro de 2019. 9

[35] Baraiya, V E V Singh: *Netflix conductor: A microservices orchestrator*, 2017. `https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40`, Acessado em 21 de fevereiro de 2019. 9

[36] Käppler, Matthias: *Inside a soundcloud microservice*, 2017. `https://developers.soundcloud.com/blog/inside-a-soundcloud-microservice`, Acessado em 21 de fevereiro de 2019. 9

[37] Haddad, Einas: *Service-oriented architecture: Scaling the uber engineering codebase we grow*, 2017. `https://eng.uber.com/soa/`, Acessado em 21 de fevereiro de 2019. 9

[38] Newman, Sam: *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015. 9

[39] PostgeSQL, 2019. `https://www.postgresql.org/`, Acessado em 14/03/2019. 14

[40] Junqueira, Flavio e Benjamin Reed: *ZooKeeper: distributed process coordination*. " O'Reilly Media, Inc.", 2013. 15

[41] ZooKeper, 2019. `https://zookeeper.apache.org/`, Acessado em 05/04/2019. 15

[42] Richardson, Leonard e Sam Ruby: *RESTful web services*. " O'Reilly Media, Inc.", 2008. 15

[43] NASA: *Nasa*, 2019. `https://www.nasa.gov/`, Acessado em 9 de junho de 2019. 19

[44] NASA: *Rackspace: Managed dedicated cloud computing services*, 2019. `https://www.rackspace.com/pt-br`, Acessado em 9 de junho de 2019. 19

[45] Rosado, Tiago e Jorge Bernardino: *An overview of openstack architecture*. Em *Proceedings of the 18th International Database Engineering & Applications Symposium*, páginas 366–367. ACM, 2014. 19

[46] OpenStack: *Openstack releases: Stein*, 2019. `https://releases.openstack.org/stein/index.html`, Acessado em 18 de junho de 2019. 19

[47] OpenStack: *Openstack releases: Train*, 2019. `https://releases.openstack.org/train/index.html`, Acessado em 18 de junho de 2019. 19

[48] OpenStack: *Openstack docs: Horizon: The openstack dashboard project*, 2019. `https://docs.openstack.org/horizon/latest/`, Acessado em 18 de junho de 2019. 20, 23

[49] Linux, Red Hat Enterprise: *Openstack docs: Openstack compute (nova)*, 2019. `https://docs.openstack.org/nova/latest/`, Acessado em 12 de junho de 2019. 20, 21, 30

[50] Linux, Red Hat Enterprise: *Openstack docs: Swift*, 2019. `https://docs.openstack.org/swift/latest/`, Acessado em 18 de junho de 2019. 20, 21, 30

[51] Linux, Red Hat Enterprise: *Openstack docs: glance*, 2019. `https://docs.openstack.org/glance/latest/`, Acessado em 18 de junho de 2019. 20, 22, 30

[52] Linux, Red Hat Enterprise: *Openstack docs: Keystone*, 2019. `https://docs.openstack.org/keystone/latest/`, Acessado em 18 de junho de 2019. 20, 23, 30

[53] Linux, Red Hat Enterprise: *Openstack docs: Cinder*, 2019. `https://docs.openstack.org/cinder/stein/`, Acessado em 21 de junho de 2019. 20

[54] Linux, Red Hat Enterprise: *Openstack docs: Nova system architecture*, 2019. `https://docs.openstack.org/nova/latest/user/architecture.html`, Acessado em 19 de junho de 2019. 21

[55] Linux, Red Hat Enterprise: *Openstack docs: Glance system architecture*, 2019. `https://docs.openstack.org/glance/latest/contributor/architecture.html`, Acessado em 19 de junho de 2019. 22

[56] Linux, Red Hat Enterprise: *Openstack docs: Neutron*, 2019. https://docs.openstack.org/neutron/stein/, Acessado em 21 de junho de 2019. 22

[57] Linux, Red Hat Enterprise: *Openstack heat*, 2019. https://www.openstack.org/software/releases/stein/components/heat, Acessado em 22 de junho de 2019. 24

[58] Linux, Red Hat Enterprise: *Openstack masakari*, 2019. https://www.openstack.org/software/releases/stein/components/masakari, Acessado em 22 de junho de 2019. 24

[59] Oracle: *Java programming language*, 2019. https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html, Acessado em 11 de junho de 2019. 26

[60] Pivotal: *Spring boot*, 2019. https://spring.io/projects/spring-boot, Acessado em 22 de junho de 2019. 26

[61] Linux, Red Hat Enterprise: *Projects | the state of the octoverse*, 2019. https://octoverse.github.com/projects.html, Acessado em 11 de junho de 2019. 27

[62] ContainX: *Openstack4j - fluent openstack client for java*, 2019. http://www.openstack4j.com/, Acessado em 11 de junho de 2019. 27

[63] Apache: *Apache jclouds*, 2019. https://jclouds.apache.org/, Acessado em 11 de junho de 2019. 27

[64] GitHub: *Github*, 2019. https://github.com/, Acessado em 11 de junho de 2019. 27

[65] Beck, Kent: *Test-driven development: by example*. Addison-Wesley Professional, 2003. 28

[66] Linux, Red Hat Enterprise: *Centos project*, 2019. https://www.centos.org/, Acessado em 11 de junho de 2019. 28

[67] Linux, Red Hat Enterprise: *Packstack - rdo*, 2019. https://www.rdoproject.org/install/packstack/, Acessado em 11 de junho de 2019. 28, 33

[68] Freeman, Eric, Elisabeth Robson, Bert Bates e Kathy Sierra: *Head first design patterns*. " O'Reilly Media, Inc.", 2004. 30

[69] JUnit: *Junit 5*, 2019. https://junit.org/junit5/, Acessado em 13 de junho de 2019. 31

[70] Angelo, Pedro Gabriel Morais e Rômulo Pires Saraiva: *Execução de workflows científicos na plataforma bionimbuz*. 2018. 33

[71] Docker, 2019. https://www.docker.com/, Acessado em 24/07/2019. 39

[72] Kubernets, 2019. https://kubernetes.io/pt/, Acessado em 24/07/2019. 39

[73] Angular: *Angularjs*, 2019. `https://angularjs.org/`, Acessado em 28 de junho de 2019. 39

[74] Ember: *Ember.js*, 2019. `https://emberjs.com/`, Acessado em 28 de junho de 2019. 39

[75] Inc., Tilde: *Vue.js*, 2019. `https://vuejs.org/`, Acessado em 28 de junho de 2019. 39